

UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INGENIERÍA DE
SISTEMAS Y AUTOMÁTICA



GENERACIÓN DE MAPAS DE DISPARIDAD UTILIZANDO CUDA

PROYECTO FIN DE CARRERA
INGENIERÍA INDUSTRIAL

AUTOR: ALEJANDRO IVÁN MARTÍN CLEMENTE
TUTOR: Dr. ARTURO DE LA ESCALERA HUESO
DIRECTOR: BASAM MUSLEH LANCIS

DICIEMBRE 2009

*Dedicado a mis padres, que aunque quizás nunca comprendan
este proyecto son los que lo han hecho realidad...*

UNIVERSIDAD CARLOS III DE MADRID

RESUMEN

UNIVERSIDAD CARLOS III DE MADRID
Departamento de Ingeniería de Sistemas y Automática

Realizado por Alejandro Iván Martín Clemente

Este proyecto ha sido desarrollado como una optimización de los algoritmos tradicionales de cálculo de mapas de disparidad. Está basado en la idea propuesta y publicada por *Joe Stam* [1] sobre imágenes estéreo con CUDA.

El presente proyecto tiene por objetivo la obtención de mapas de disparidad en un sistema de visión estéreo donde el tiempo de cómputo es la variable crítica. Para ello se utilizará la tecnología innovadora llamada CUDA, que permite mediante el uso de la tarjeta gráfica paralelizar operaciones para reducir hasta en varios órdenes de magnitud los tiempos de cálculo.

Todo lo realizado apunta hacia futuras implementaciones en sistemas tales como vehículos inteligentes, donde lo que prima es el tiempo de cálculo para la toma de decisiones en tiempo real, de modo que se pueda avisar al conductor o al sistema inteligente que dirija el vehículo.

Una vez desarrollado el software, se estudiarán los parámetros necesarios de ajustar para que puedan obtenerse unos resultados óptimos, en cualquier hardware compatible con dicha tecnología.

Leganés, Diciembre de 2009

ÍNDICE GENERAL

RESUMEN..... ii

ÍNDICE DE FIGURAS vii

ÍNDICE DE TABLAS xi

LISTADO DE ABREVIATURAS xii

CAPÍTULO 1. INTRODUCCIÓN - 1 -

1.1 Estado del arte - 1 -

1.1.1 El vehículo experimental IVVI - 2 -

1.1.2 iCab - 5 -

1.2 Introducción a la visión estéreo..... - 5 -

1.3 Objetivos..... - 7 -

CAPÍTULO 2. ASPECTOS TEÓRICOS - 8 -

2.1 Conceptos previos - 8 -

2.1.1 Modelo pin-hole - 8 -

2.1.2 Sistema compuesto por dos cámaras iguales y paralelas (no real en la práctica) - 10 -

2.2 Visión estéreo - 14 -

2.2.1 Geometría epipolar - 14 -

2.2.1.1	Matriz esencial	- 17 -
2.2.1.2	Matriz fundamental	- 19 -
2.2.1.3	Cálculo de la matriz fundamental o la matriz esencial.....	- 22 -
2.2.2	Calibración	- 22 -
2.2.2.1	Modelo de cámara en coordenadas homogéneas.....	- 24 -
2.2.2.2	Calibración pasiva	- 27 -
2.2.2.3	Calibración activa	- 29 -
2.2.2.4	Calibración de la cámara por sí misma.....	- 30 -
2.2.2.5	Rectificación	- 30 -
2.2.3	Correspondencia	- 32 -
2.2.3.1	Correlación	- 33 -
2.2.3.2	Obtención del máximo y de la disparidad.....	- 34 -
2.2.4	Filtrado de la imagen de disparidades	- 36 -
2.2.5	Problemas computacionales de la visión estéreo.....	- 37 -
CAPÍTULO 3. INTRODUCCIÓN A NVIDIA® CUDA™.....		- 39 -
3.1	Un modelo de programación paralela y escalable	- 39 -
3.2	Ventajas del uso de la GPU	- 39 -
3.3	Modelo de programación en CUDA.....	- 42 -
3.4	Jerarquía de memoria.....	- 45 -
3.5	Modelo de ejecución	- 46 -
3.6	Host (CPU) y Device (GPU)	- 46 -

3.7	Hardware GPU	- 48 -
3.8	Compilar aplicaciones CUDA.....	- 49 -
3.9	Consejos para la programación con CUDA	- 50 -
3.10	Justificación de la elección de NVIDIA CUDA como entorno de trabajo	- 52 -
CAPÍTULO 4. DESARROLLO DEL PROYECTO		- 53 -
4.1	Código host (CPU)	- 53 -
4.2	Código device (GPU): kernel para la aplicación de la Laplaciana de la Gausiana	- 54 -
4.2.1	Eliminación de ruido en una imagen.....	- 55 -
4.2.2	Detección de bordes en una imagen	- 58 -
4.2.3	Justificación del uso de la Laplaciana de la Gausiana y del método SSD	- 59 -
4.2.4	Justificación del algoritmo implementado.....	- 60 -
4.2.5	Interpretación de los resultados	- 62 -
4.2.6	Aplicación a dos imágenes	- 63 -
4.3	Código device (GPU): algoritmo para la obtención del mapa de disparidades	- 64 -
4.3.1	Procedimiento detallado	- 65 -
4.3.2	Justificación de la solución adoptada	- 69 -
CAPÍTULO 5. RESULTADOS		- 71 -
5.1	Gráficas de tiempos de cómputo.....	- 71 -
5.1.1	Interpretación de las gráficas.....	- 73 -
5.1.1.1	Coste computacional frente a la máxima disparidad de búsqueda	- 73 -
5.1.1.2	Coste computacional frente a la dimensión de la ventana de búsqueda	- 74 -

5.2	Mapas de disparidades obtenidos	- 76 -
5.2.1	Comentarios acerca de los resultados	- 78 -
CAPÍTULO 6. CONCLUSIONES Y TRABAJOS FUTUROS		- 80 -
6.1	Cumplimiento de objetivos.....	- 80 -
6.2	Implementación en un sistema real	- 80 -
6.3	Líneas futuras de investigación.....	- 83 -
CAPÍTULO 7. COSTES DEL PROYECTO		- 87 -
APÉNDICE A. MANUAL DE USUARIO		- 89 -
A.1	Opciones modificables del código host (CPU)	- 89 -
A.2	Opciones modificables del código device (GPU)	- 89 -
APÉNDICE B. SCRIPT DE MATLAB.....		- 95 -
APÉNDICE C. CÓDIGO HOST.....		- 97 -
APÉNDICE D. CÓDIGO DEVICE.....		- 104 -
APÉNDICE E. CARACTERÍSTICAS DE LA GPU.....		- 117 -
REFERENCIAS BIBLIOGRÁFICAS		- 119 -

ÍNDICE DE FIGURAS

Figura 1.1	Vehículo IVVI. (a) Vista general (b) Cámara B&W perteneciente al sistema estéreo y cámara color (c) Espacio para el operador (d) GPS y sistema de comunicación.	- 2 -
Figura 1.2	Imagen tomada por una de las cámaras del vehículo IVVI.....	- 3 -
Figura 1.3	Cámara de evaluación de la atención del conductor con iluminación infrarroja.....	- 4 -
Figura 1.4	Cámara de infrarrojo lejano de captación de calor emitido por los objetos.....	- 4 -
Figura 1.5	iCab.....	- 5 -
Figura 2.1	Modelo de lente fina.	- 8 -
Figura 2.2	Cámara de tipo pin-hole.....	- 9 -
Figura 2.3	Modelo pin-hole con imagen formada invertida y sin invertir.....	- 10 -
Figura 2.4	Modelo pin-hole para el caso de dos cámaras paralelas.	- 11 -
Figura 2.5	Variación de la resolución y del ángulo de visión con la distancia focal y la separación entre cámaras.	- 12 -
Figura 2.6	Distancia en metros frente a la disparidad en píxeles para la cámara <i>Point Grey Research Mumblebee2</i>	- 13 -
Figura 2.7	Geometría epipolar.	- 15 -
Figura 2.8	Localización de las líneas epipolares para cámaras no paralelas. Situación de las cámaras (a). Imágenes tomadas por ambas cámaras así como las líneas epipolares conjugadas (b)(c).	- 21 -
Figura 2.9	Localización de las líneas epipolares para cámaras paralelas. Situación de las cámaras (a). Imágenes tomadas por ambas cámaras así como las líneas epipolares conjugadas (b) (c).	- 22 -

Figura 2.10 Sistema estéreo no calibrado. Imágenes izquierda y derecha originales (a) (b). Mapa de disparidades entre ambas imágenes (c).....	23 -
Figura 2.11 Rectificación.	24 -
Figura 2.12 Modelo de cámara en coordenadas homogéneas.	25 -
Figura 2.13 Posición de las líneas epipolares para distintos casos.	27 -
Figura 2.14 Patrón de calibración.	28 -
Figura 2.15 Sistemas de coordenadas.....	29 -
Figura 2.16 Resultados. Imágenes izquierda y derecha sin rectificar (a) (b). Imágenes izquierda y derecha rectificadas (c) (d). Mapa de disparidades entre a y b (e). Mapa de disparidades entre c y d (f).....	31 -
Figura 2.17 Disparidad con precisión de píxel.....	35 -
Figura 2.18 Disparidad con precisión subpíxel.	35 -
Figura 2.19 Imágenes izquierda y derecha (a) (b). Mapa de disparidades sin filtrar (c) (d). Mapa de disparidades filtrado (e) (f).....	38 -
Figura 3.1 Comparación evolutiva entre los GFLOP/s para la CPU y la GPU.	40 -
Figura 3.2 Comparación evolutiva entre el ancho de banda (GB/s) para la CPU y la GPU.	40 -
Figura 3.3 La GPU dedica más transistores al procesamiento de datos.	41 -
Figura 3.4 Tarjeta gráfica NVIDIA® GeForce® GTX 295 compatible con la tecnología CUDA.	42 -
Figura 3.5 Grid de bloques de hilos.....	43 -
Figura 3.6 Jerarquía de memoria.	45 -
Figura 3.7 Programación heterogénea. El código se ejecuta en serie en el <i>device</i> mientras que en el <i>host</i> se ejecuta en paralelo.....	47 -
Figura 3.8 Modelo de Hardware de la GPU.....	49 -
Figura 3.9 Patrón de acceso a memoria compartida con conflictos (a) y sin conflictos (b).	51 -

Figura 4.1	Imagen original (a). Ruido gaussiano (b).....	- 55 -
Figura 4.2	Imagen original (a). Ruido impulsional (b).	- 56 -
Figura 4.3	Imagen original (a). Ruido frecuencial (b).	- 56 -
Figura 4.4	Imagen original (a). Ruido multiplicativo (b).	- 57 -
Figura 4.5	Resultado de la aplicación del kernel de la Laplaciana de la Gaussiana. Imagen original (a). Laplaciana de la Gaussiana (b).....	- 64 -
Figura 4.6	Correlación izquierda-derecha. Imagen de referencia(a). Imagen donde se busca (b).	- 65 -
Figura 4.7	Esquema del proceso implementado utilizando una ventana de búsqueda de 5x5 píxeles y un tamaño del bloque de hilos de (16, 1, 1) píxeles.	- 68 -
Figura 5.1	Gráficas de tiempos de cómputo en función de la máxima disparidad de búsqueda (a) y el tamaño de la ventana de búsqueda (b). Ambas realizadas con NVIDIA ® GeForce® 8600M GT y la configuración óptima del algoritmo para dos imágenes de 640x480 píxeles.....	- 72 -
Figura 5.2	Imágenes originales de entrada al algoritmo ambas de resolución 640x480 píxeles. Imagen izquierda ya rectificada (a). Imagen derecha ya rectificada (b).	- 76 -
Figura 5.3	Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 7x7 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 155msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA ® GeForce® 8600M GT.....	- 77 -
Figura 5.4	Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 11x11 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 169msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA ® GeForce® 8600M GT.	- 77 -
Figura 5.5	Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 17x17 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 189msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA ® GeForce® 8600M GT.	- 78 -

Figura 5.6	Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 7x7 píxeles y una limitación de máxima disparidad de 150 píxeles. El tiempo de cómputo fue de 272msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA® GeForce® 8600M GT.....	- 78 -
Figura 6.1	Vista lateral del IVVI 2.0.	- 80 -
Figura 6.2	Vista frontal del IVVI 2.0.....	- 81 -
Figura 6.3	Sistema estéreo del IVVI 2.0.....	- 81 -
Figura 6.4	Ordenadores Apple situados en el maletero del IVVI 2.0.	- 82 -
Figura 6.5	Monitor de control en el IVVI 2.0.....	- 82 -
Figura 6.6	Par de imágenes estéreo rectificadas.	- 84 -
Figura 6.7	En esta imagen el tono de cada píxel indica el tamaño utilizado para la ventana de búsqueda (tonos más claros indican un mayor tamaño de ventana).	- 84 -
Figura 6.8	Mapa de disparidades con precisión de píxel (a). Mapa de disparidades con precisión subpíxel y con falso color (b).	- 85 -
Figura A.1	Consola de resultados de ejecución del programa.	- 94 -

ÍNDICE DE TABLAS

Tabla 2.1	Valores de calibración	- 29 -
Tabla 4.1	Comparativa de tiempos entre ambos métodos, utilizando una GPU NVIDIA® GeForce® 8600M GT y un tamaño de imagen de 640x480 píxeles (VGA).....	- 61 -
Tabla 4.2	Comparativa de tiempos entre ambos métodos para dos imágenes de 640x480 píxeles, utilizando una GPU NVIDIA® GeForce® 8600M GT.	- 63 -
Tabla 7.1	Tiempo invertido en cada fase del proyecto.	- 87 -
Tabla 7.2	Costes económicos del proyecto.....	- 88 -

LISTADO DE ABREVIATURAS

ADAS	A dvanced D river A ssistance S ystems
CCD	C harge- C oupled D evice
CPU	C entral P rocessing U nit
CUDA	C ompute U nified D evice A rchitecture
FLOPS	F loating point O perations P er S econd
FPS	F rames P er S econd
GPU	G raphic P rocessor U nit
iCab	I ntelligent C ampus automo B ile
IVVI	I ntelligent V ehicle based on V isual I nformation
PGM	P ortable G ray M ap
SP	S calar P rocessor

CAPÍTULO 1. INTRODUCCIÓN

1.1 Estado del arte

Es bastante conocido el hecho de que los errores humanos son los causantes de la mayoría de accidentes de tráfico [2]. Los dos principales errores son la distracción y la toma de decisiones incorrectas del conductor. En estos últimos años se están intentando reducir los accidentes por medio de campañas, educación preventiva, etc. pero hay una parte remanente de ellos que es muy difícil, o prácticamente imposible de eliminar. Es en este punto donde cabe nombrar la inclusión de ADAS en los vehículos, ya que son los que en realidad pueden reducir de forma drástica el número, peligro y severidad de los accidentes de tráfico.

La visión por computador juega un importante papel en este ámbito, debido a la gran cantidad de información visual que se recibe. Por ejemplo, la tarea de conducción es la que tiene en cuenta la información visual para evitar obstáculos, mientras se conduce por el sitio correcto y a la velocidad adecuada.

El resto de información visual, como la detección de señales y reconocimiento de las mismas, está relacionado con el éxito de la conducción, y con la llegada al destino, lo cual es el objetivo principal del conductor. Los ADAS se diseñan para ayudar a los conductores humanos en estas tareas. De este modo, se encuentran ejemplos de detección y seguimiento de marcas viales, detección de obstáculos en la trayectoria del vehículo (peatones, vehículos, semáforos, marcas en carreteras, etc.). Con ADAS la tarea de identificar todos estos objetos de interés ayudaría enormemente a los conductores humanos, permitiéndoles aumentar su concentración en la conducción del vehículo.

Por todas las razones nombradas anteriormente, ADAS basados en visión por Computador pueden ayudar a los conductores humanos de muchas maneras, como por ejemplo:

- *Protección de peatones.*
- *Control adaptativo de crucero.* Para calcular la velocidad óptima del vehículo en función de parámetros como las trayectorias y velocidades de los otros vehículos.
- *Reconocimiento de señales de tráfico.*

En la Figura 1.1 se puede observar un vehículo dotado de sistemas ADAS.

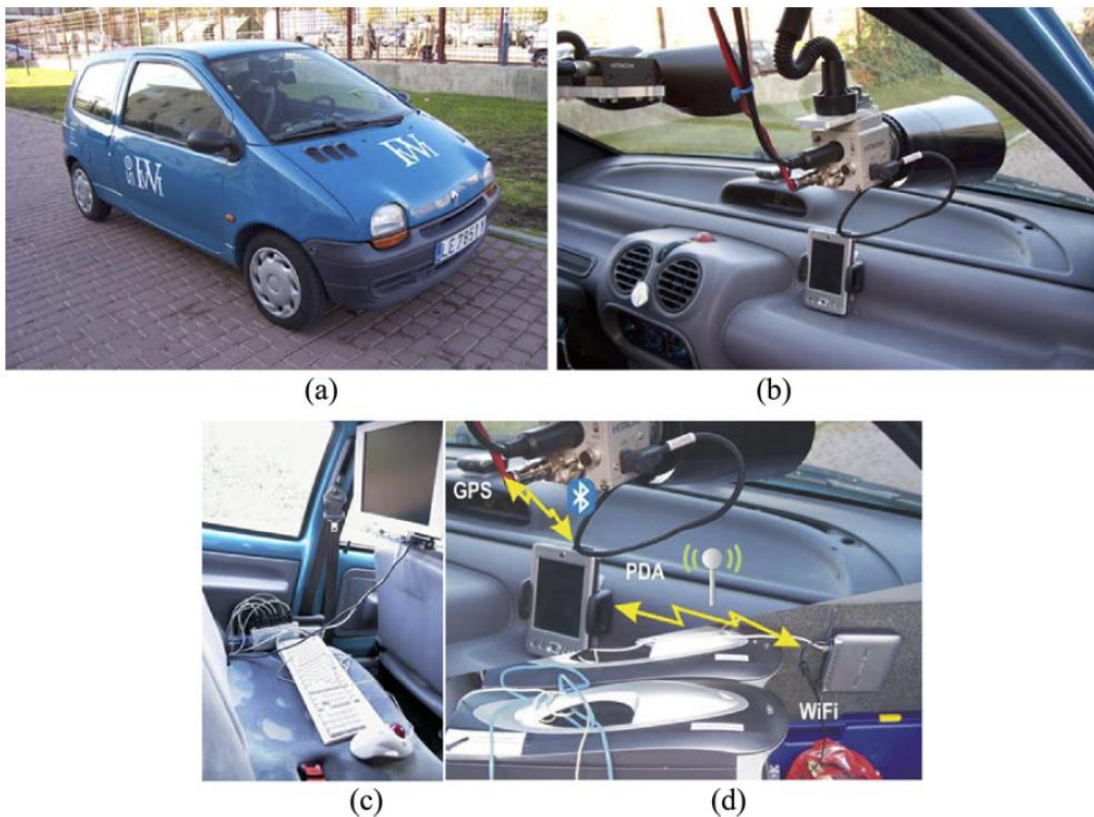


Figura 1.1 Vehículo IVVI. (a) Vista general (b) Cámara B&W perteneciente al sistema estéreo y cámara color (c) Espacio para el operador (d) GPS y sistema de comunicación.

1.1.1 El vehículo experimental IVVI

IVVI (Figura 1.1 (a)) es un vehículo de investigación para la implementación de sistemas basados en visión por computador, con el objetivo de implementar ADAS. El propósito de la plataforma IVVI es evaluar los algoritmos de visión por computador bajo condiciones reales de funcionamiento. Todos los ADAS diseñados funcionan en ambientes urbanos y en carretera, suministrando avisos en tiempo real al conductor.

En la Figura 1.1 (b) se puede observar una de las cámaras pertenecientes al sistema de visión estéreo binocular. Además hay una cámara CCD para la detección de señales de tráfico y otras señales verticales. La velocidad del vehículo, posición y trayectoria se obtienen mediante un GPS conectado a una PDA por Bluetooth. La PDA a su vez transmite la información a los PCs a través de un router WiFi (Figura 1.1 (d)).

Mientras uno de los ordenadores recibe la información del sistema estéreo y almacena los resultados, vehículos y detección de peatones; el segundo ordenador analiza la información de las señales de tráfico detectadas. De este modo un ordenador previene la colisión entre coches, mientras que el otro atiende a las infracciones de tráfico. Como forma de comunicación con el conductor, ambos ordenadores están conectados a los altavoces del vehículo para poder realizar las advertencias. En caso de conflicto, se tienen priorizados una serie de mensajes por orden de peligrosidad previamente definidos.

Como se puede observar en la Figura 1.2, esto sería lo capturado por una de las cámaras de las que dispone dicho vehículo, de la cual se podrán extraer características como la detección de líneas de la carretera. Principalmente los algoritmos programados están pensados para funcionar en vías con marcas viales visibles, que se podrán detectar mediante diversas técnicas de visión por computador para ayudar en la conducción.



Figura 1.2 Imagen tomada por una de las cámaras del vehículo IVVI.

Además de los sistemas nombrados anteriormente, también dispone de una cámara que evalúa la atención del conductor, hasta en condiciones de poca luz, mediante iluminación infrarroja (Figura 1.3).



Figura 1.3 Cámara de evaluación de la atención del conductor con iluminación infrarroja.

También dispone de otra cámara infrarroja, que percibe el calor emitido por los objetos, de esta manera los peatones pueden ser detectados en condiciones de baja visibilidad (Figura 1.4).

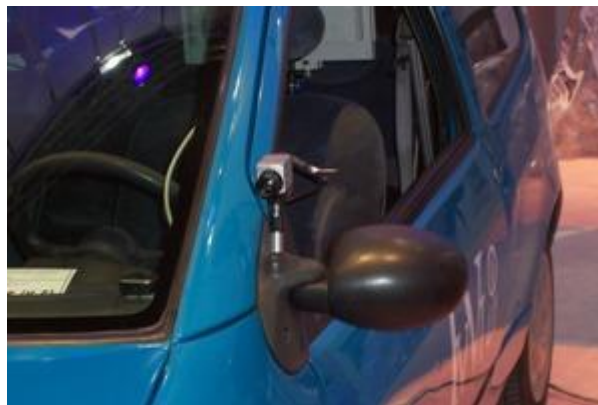


Figura 1.4 Cámara de infrarrojo lejano de captación de calor emitido por los objetos.

1.1.2 iCab

Se trata de un vehículo eléctrico, en el que se ha modificado la electrónica y la mecánica para que pueda ser controlado por un ordenador embarcado.

Su función será llevar a los visitantes del Campus de forma autónoma de una zona a otra, para lo que se dispondrá de una flota de vehículos (Figura 1.5).



Figura 1.5 iCab.

1.2 Introducción a la visión estéreo

Una imagen es una proyección bidimensional de un mundo tridimensional [3]. Por ello no se puede obtener información de distancias a menos que se introduzcan restricciones, bien de la posición de los objetos (caso de la iluminación estructurada), o bien se conozcan sus dimensiones reales. La solución *natural* es utilizar es utilizar dos cámaras (visión estéreo o estereoscópica), pero entonces se presenta el problema que plantea el alto coste computacional requerido para su cálculo (en tiempos relativamente bajos). Por este motivo se han utilizado otros sistemas como láseres, que sí que dan una imagen de distancias directamente, con un tiempo de cómputo relativamente bajo.

La visión estéreo, por el contrario, es una técnica muy poco utilizada, pero muy potente para determinar la distancia a los objetos, utilizando un par de cámaras

previamente calibradas y alineadas. Este procedimiento es el mismo sistema visual que el usado por los humanos y la gran mayoría de animales. El principal problema que hasta la actualidad que ha presentado la visión estéreo, es debido al alto coste computacional. Por ello otros sistemas como los nombrados anteriormente, son los que se utilizan para aplicaciones en tiempo real y éste se suele orientar a aquellas, en las cuales se dispone de una potencia computacional elevada.

Sin embargo, la visión estéreo presenta una serie de ventajas sobre otros sistemas:

- Es un sistema no invasivo, no hay que emitir ninguna forma de energía (luz o radio).
- Puede producir imágenes “densas” y precisas de distancias.
- Se dispone de dos tipos de información (visual y distancias) con un único sensor.
- No sufre de las deformaciones producidas por el movimiento de los vehículos mientras el sensor rastrea la zona de trabajo, siempre y cuando las cámaras estén sincronizadas y el tiempo de exposición sea bajo.

Las aplicaciones de un sistema estéreo son varias:

- Robótica: mapas de elevación para la navegación de un robot móvil, detección tridimensional de piezas, etc.
- Modelado de piezas: Obtención de modelos 3D para gráficos por ordenador, para sistemas CAD.
- Teleconferencia: Inserción de imágenes en modelos virtuales.

Para una aplicación típica, el par estéreo estará separado, más o menos, la distancia de los ojos humanos. Cada cámara capturará la imagen desde una posición diferente, y los objetos serán capturados desde un ángulo ligeramente distinto. Como consecuencia, para los objetos cercanos, la diferencia de ángulo será mayor que para los objetos más lejanos.

1.3 Objetivos

El principal objetivo de este proyecto, es la obtención de mapas de disparidad a partir de dos imágenes estéreo, minimizando el tiempo de cómputo global. Para ello se tratará de desarrollar un método simple y efectivo, para determinar la disparidad entre píxeles de dos imágenes (tarea más costosa en la elaboración del mapa de disparidad), usando una técnica de *matching* para determinar la correlación entre grupos de píxeles de un par estéreo.

Otro objetivo también importante es la flexibilidad del código implementado, es decir, que se permita el cambio de parámetros de forma sencilla, para que se eviten problemas a la hora de introducirse en otro software más extenso.

Una vez planteados los objetivos, también es importante conocer los problemas que se pueden plantear, como:

- Iluminación no controlada en el entorno de trabajo.
- Aparición de zonas con poca textura, lo cual es uno de los problemas más importantes en la visión estereoscópica.
- El tiempo de cómputo. Limitará la configuración del algoritmo así como su complejidad.

Una vez establecidos los objetivos del proyecto, a continuación se desarrollarán algunos conceptos teóricos de visión estereoscópica, necesarios para llevar a cabo el desarrollo del mismo.

CAPÍTULO 2. ASPECTOS TEÓRICOS

2.1 Conceptos previos

Es necesario definir conceptos como el modelo pin-hole de la cámara, entre otros, ya que se utilizarán estos conceptos en desarrollos posteriores.

2.1.1 Modelo pin-hole

La función de la óptica de una cámara es captar los rayos luminosos y concentrarlos sobre el sensor de la misma. Cuando los rayos paralelos pasan a través de una lente convexa, convergen hacia un punto que se denomina *punto focal* (o *foco*).

El *eje óptico* es la línea perpendicular a la lente y que la atraviesa por su punto medio.

La distancia entre el eje de la lente y el punto focal se denomina *distancia focal* (f). Este parámetro es fundamental a la hora de calcular la posición y el tamaño de los objetos en la imagen.

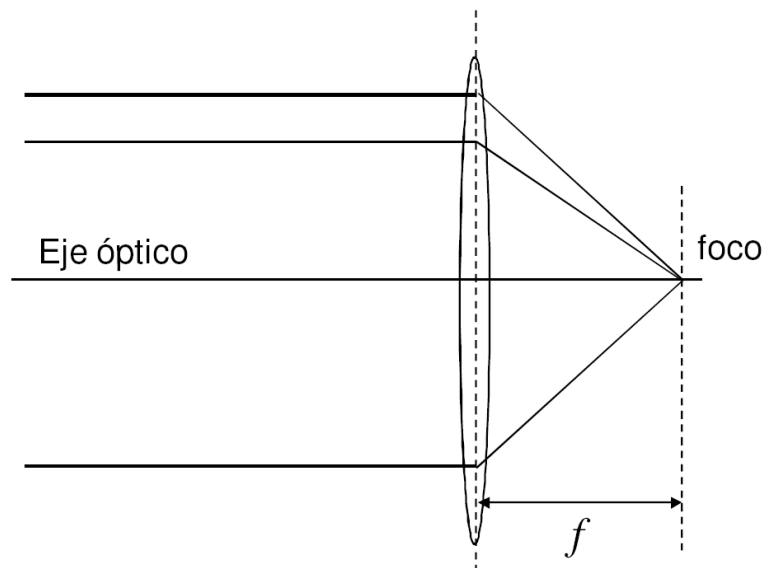


Figura 2.1 Modelo de lente fina.

Una vez se han definido estos conceptos básicos de toda óptica, se puede plantear un primer modelo de lente, que se denominará de la *lente fina* (Figura 2.1). En este modelo, los rayos que inciden paralelamente al eje óptico convergen en el foco.

Otro de los modelos que intenta reflejar el comportamiento de la óptica, es el denominado *pin-hole*, que destaca por su sencillez. Este modelo se basa en reducir la óptica a un punto situado a la distancia focal de la imagen [4]. De todos los rayos luminosos que refleja un punto perteneciente a un objeto, solamente se considera el que pasa directamente por la distancia focal. Además, este modelo supone que todos los puntos están enfocados y no se tienen en cuenta las imperfecciones que introducen en las imágenes los sistemas ópticos.

Este modelo es el que usaban las primerísimas cámaras, las cuales estaban constituidas por una caja, en la que se realizaba una apertura muy pequeña en un lado y se colocaba la película en el otro (Figura 2.2).

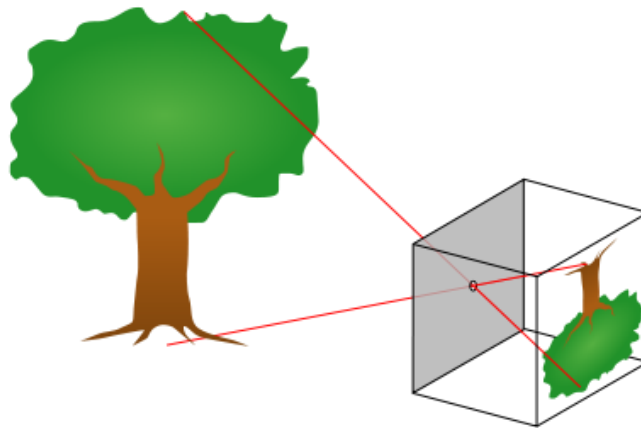


Figura 2.2 Cámara de tipo pin-hole.

De forma matemática se puede expresar este modelo como se observa en la Figura 2.3, donde es fácil, por triangulación, ver que las ecuaciones que relacionan un punto del mundo con su proyección en la imagen son:

$$\begin{aligned} x &= \frac{f}{Z} X \\ y &= \frac{f}{Z} Y \end{aligned} \quad (2.1)$$

Según estas expresiones, es evidente la pérdida de una dimensión (la profundidad Z). Por otro lado, como también puede verse en la Figura 2.3, la imagen formada está invertida, por lo que habrá que cambiar los sentidos de los ejes. Una posibilidad para solucionar este problema, es suponer que el sensor se encuentra antes del punto focal, lo que también se observa en la misma figura.

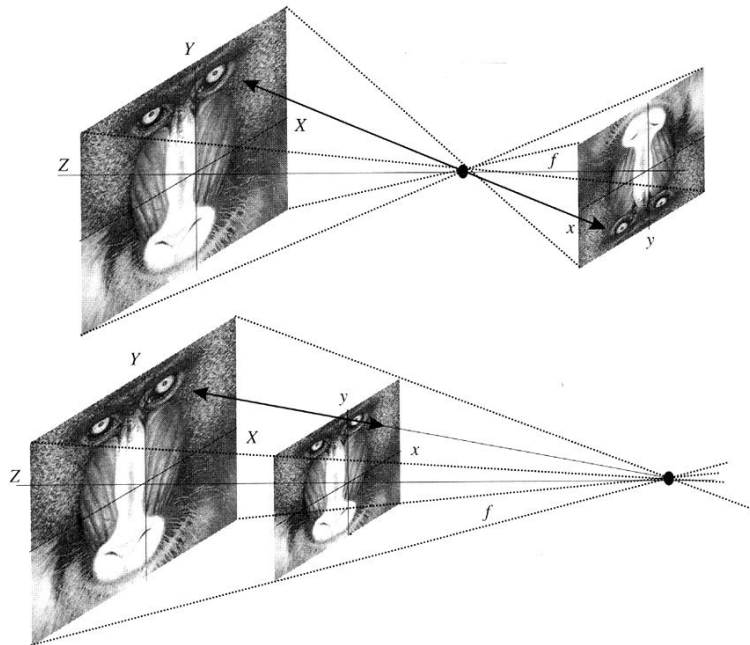


Figura 2.3 Modelo pin-hole con imagen formada invertida y sin invertir.

2.1.2 Sistema compuesto por dos cámaras iguales y paralelas (no real en la práctica)

Si se consideran dos cámaras de igual distancia focal f , paralelas y separadas una distancia d , como se expresa en [3], el mismo punto P se ve proyectado en las dos imágenes en los píxeles $p_1 (u_1, v_1)$ y $p_2 (u_2, v_2)$. En general las coordenadas de estos dos puntos no serán iguales, denominándose *disparidad* a la diferencia entre ambos.

La estructura básica de un sistema de visión estéreo simplificado al modelo pin-hole es la que se muestra en la Figura 2.4.

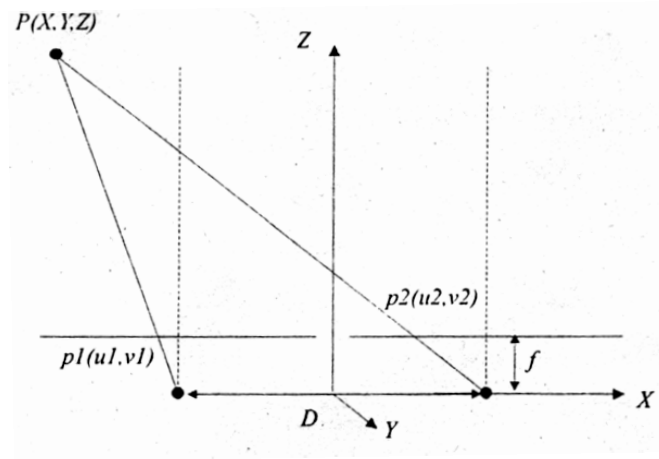


Figura 2.4 Modelo pin-hole para el caso de dos cámaras paralelas.

Las ecuaciones que se pueden plantear para cada una de las cámaras son (según el modelo pin-hole):

$$\frac{u_1}{f} = \frac{X + D/2}{Z}$$

$$\frac{u_2}{f} = \frac{X - D/2}{Z}$$

Donde X y Z son las incógnitas y el resto son datos. Si se igualan las coordenadas horizontales se llega a:

$$X = Z \cdot \frac{u_1}{f} - \frac{D}{2} = Z \cdot \frac{u_2}{f} + \frac{D}{2} \quad (2.2)$$

$$Z = f \cdot \frac{D}{u_1 - u_2} = f \cdot \frac{D}{d}$$

Siendo d la disparidad, que será siempre positiva y D la distancia entre las cámaras (denominada *baseline*). La coordenada Y por lo tanto será:

$$\frac{v_1}{f} = \frac{Y}{Z} \Rightarrow Y = \frac{v_1}{f} \cdot \frac{D \cdot f}{d} = \frac{D \cdot v_1}{d} \quad (2.3)$$

Si se compara con el resultado obtenido con la otra cámara se llega a:

$$\frac{v_2}{f} = \frac{Y}{Z} = \frac{D \cdot v_1}{d} \cdot \frac{d}{D \cdot f} = \frac{v_1}{f} \quad (2.4)$$

En este caso se ve claramente que v_1 es igual a v_2 . Luego si se toma un punto de la cámara izquierda, su correspondiente en la imagen derecha estará a la misma altura. La línea recta donde se encuentran estos puntos, que son los posibles candidatos, se denomina *línea epipolar*, y para el caso que se está estudiando en este apartado serán siempre líneas paralelas horizontales.

De la ecuación 2.2 puede derivarse que si las cámaras están más separadas, o la distancia focal es mayor, se tiene una mejor resolución, pero en contrapartida peor ángulo de visión. En la Figura 2.5 se puede observar este efecto.

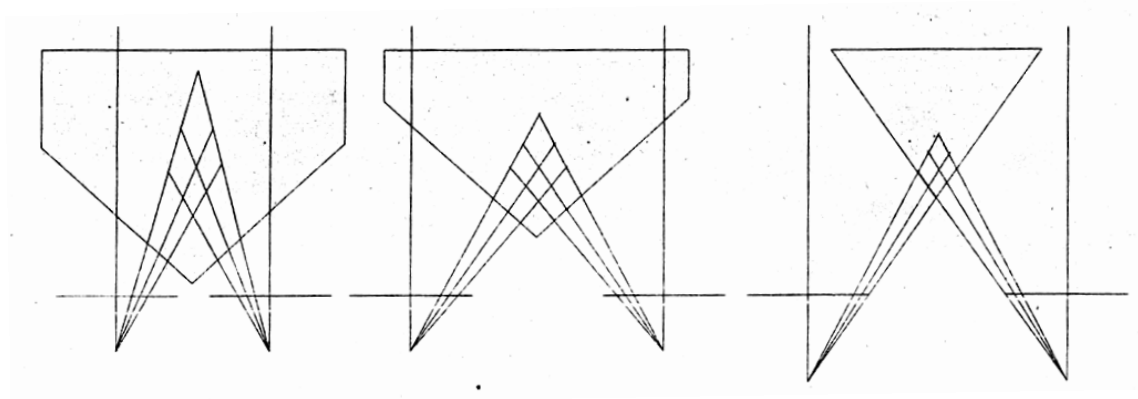


Figura 2.5 Variación de la resolución y del ángulo de visión con la distancia focal y la separación entre cámaras.

Como se deduce de las expresiones anteriores, el principal problema a resolver es el cálculo de la disparidad d (medida en píxeles en el algoritmo implementado), la cual se puede convertir a unidades de distancia multiplicando por el tamaño de cada píxel, para obtener el valor d . Si este valor d se normaliza (d'), como es nuestro caso, para que la salida de la aplicación sea una imagen normalizada con valores entre [0 255] la distancia será:

$$Z = f \cdot \frac{D \cdot 255}{T \cdot d' \cdot B} \quad (2.5)$$

Donde T es el tamaño del píxel en el eje x , B es el valor de restricción de máxima disparidad del algoritmo que se explicará más adelante y d' el valor de disparidad normalizado de [0 255] de salida de la aplicación.

Como se observa, todo son constantes excepto el valor de d , por lo cual se puede reducir a:

$$Z(\text{metros}) = \frac{k}{d(\text{píxeles})} \quad (2.6)$$

En la Figura 2.6 se observa la relación entre la disparidad y la distancia a los objetos. Cabe destacar que las distancias están en escala logarítmica. Esta gráfica se realizó usando los parámetros de una cámara estéreo *Point Grey Research Mumblebee2* con una distancia entre cámaras (D) de 120mm, una distancia focal (f) de 3.8mm y un píxel pitch de 4.65 μm

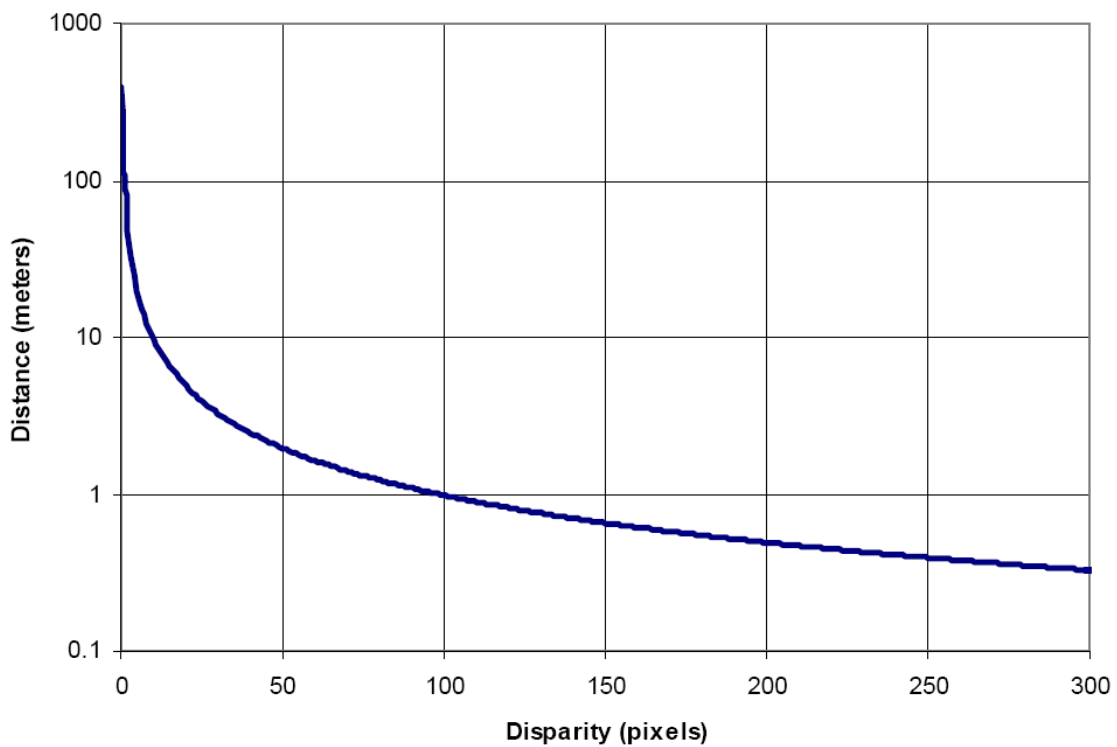


Figura 2.6 Distancia en metros frente a la disparidad en píxeles para la cámara *Point Grey Research Mumblebee2*.

Como se observa en el gráfico, para objetos cercanos la distancia varía poco con las variaciones de disparidad, mientras que para objetos lejanos la distancia varía

enormemente para pequeñas variaciones de disparidad. Como consecuencia, para detectar objetos lejanos se requerirá una alta resolución y un cálculo de disparidades con alta precisión. Para objetos cercanos, sin embargo, no se requerirá de tanta precisión.

2.2 Visión estéreo

Una vez se ha explicado, a modo introductorio, la parte teórica de un sistema estéreo, se pasarán a definir los pasos de todo sistema de visión estéreo completo para su puesta en funcionamiento [5]:

- Calibración:
 - Corrección geométrica.
 - Transformación de la imagen.
- Correspondencia:
 - Correlación.
 - Obtención del máximo y de la disparidad.
- Filtrado de la imagen de disparidades

Antes de comenzar con el apartado de calibración de las cámaras, se estudiará más a fondo la geometría epipolar, propia de un sistema de visión estéreo.

2.2.1 Geometría epipolar

La geometría de la visión estéreo se denomina epipolar [6] y sigue el esquema mostrado en la Figura 2.7, donde se pueden apreciar dos cámaras de tipo pin-hole, sus centros de proyección, O_l y O_r , y sus planos imagen o retinas en coordenadas normalizadas. Las longitudes focales se denotan como f_l y f_d .

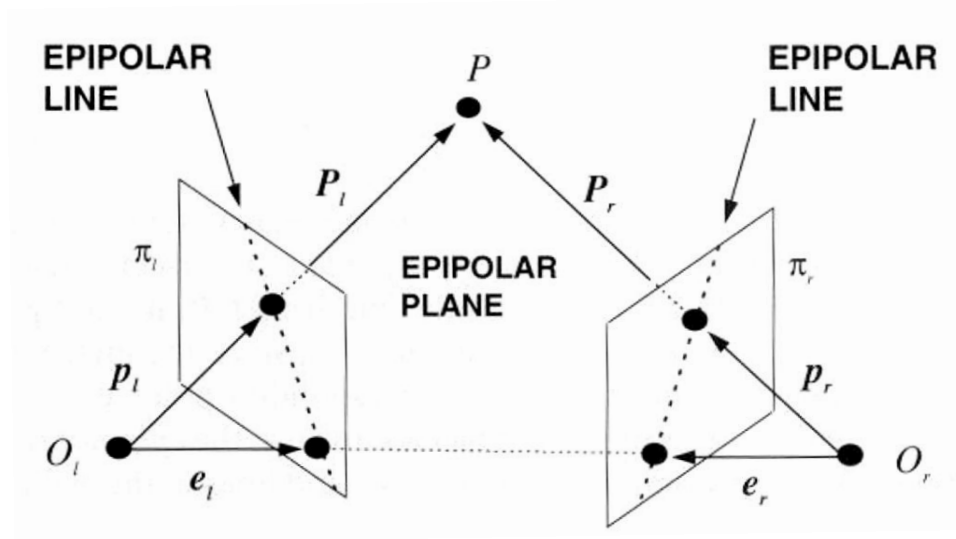


Figura 2.7 Geometría epipolar.

Cada cámara identifica un sistema de referencia 3D, el origen del cual se sitúa en el centro de proyección de la cámara y el eje Z con el eje óptico. Los vectores denotados como $P_l = [X_l, Y_l, Z_l]^T$ y $P_r = [X_r, Y_r, Z_r]^T$ se refieren al mismo punto 3D P , como vectores en los sistemas de referencia de la cámara izquierda y derecha respectivamente. Los vectores $p_l = [x_l, y_l, z_l]^T$ y $p_r = [x_r, y_r, z_r]^T$ definen las proyecciones del punto P , en la imagen izquierda y derecha respectivamente, y están expresados en su correspondiente sistema de referencia. Por otra parte, es trivial que para todos los puntos de las imágenes tenemos $z_l = f_l$ ó $z_r = f_r$ de acuerdo a la imagen que sea.

Los sistemas de referencia de las cámaras izquierda y derecha están relacionados a través de los parámetros extrínsecos (los cuales describen la relación espacial entre la cámara y el mundo). Estos parámetros definen una transformación rígida en el espacio 3D, definida por un vector de traslación $T = O_r - O_l$ y una matriz de rotación R . Dado un punto P en el espacio, la relación entre P_l y P_r es por tanto:

$$P_r = R(P_l - T) \quad (2.7)$$

Los puntos de corte de la recta que une los centros de proyección de ambas cámaras, con los planos de proyección se llaman epipolos, de ahí el nombre de geometría epipolar.

Se denotará por e_l y e_r el epipolo izquierdo y derecho respectivamente. Por construcción, ambos epipolos representan la proyección, en su correspondiente plano, de la imagen del centro de proyección de la otra cámara. En el caso de que uno de los planos de imagen sea paralelo a la recta que une los centros de proyección, el epipolo de ese plano estará situado en el infinito.

La relación entre un punto del espacio 3D y su proyección en las imágenes se describe por las siguientes ecuaciones:

$$\begin{aligned} p_l &= \frac{f_l}{Z_l} P_l \\ p_r &= \frac{f_r}{Z_r} P_r \end{aligned} \tag{2.8}$$

La importancia práctica de la geometría epipolar arranca del hecho que el plano identificado por P, O_l, O_r , llamado plano epipolar, interseca cada imagen en una línea llamada línea epipolar. Considérense ahora los puntos P, p_l y p_r . Dado p_l, P puede caer en cualquier punto del rayo definido por O_l y p_l . Pero dado que la imagen de este rayo en la imagen derecha es la línea epipolar a través del punto correspondiente p_r dicho punto debe estar sobre la línea epipolar. Esta correspondencia establece una aplicación entre puntos de la imagen izquierda y rectas de la imagen derecha (y viceversa). Una consecuencia de esta correspondencia es, que dado que todos los rayos pasan, por construcción, por el centro de proyección, todas las rectas epipolares deben pasar por el epipolo.

Por lo tanto si se determina la aplicación entre puntos de la imagen izquierda y las rectas epipolares de la imagen derecha (y viceversa), se puede restringir la búsqueda para el emparejamiento de p_l a lo largo de la línea epipolar correspondiente.

Así pues, la búsqueda de las correspondencias se reduce a un problema 1D. Alternativamente, este conocimiento también se puede usar para verificar si una potencial pareja de puntos correspondientes, lo son de verdad o no. Esta técnica es, normalmente, una de las más efectivas para detectar las posibles falsas correspondencias debidas a oclusión.

Restricción epipolar: Los puntos correspondientes deben estar sobre líneas epipolares conjugadas.

A continuación, se estudiarán técnicas para estimar la geometría epipolar, para que sea posible establecer una aplicación entre puntos de una imagen y líneas epipolares de la otra.

2.2.1.1 Matriz esencial

Seleccionando el origen de coordenadas centrado en O_l (se puede elegir el centro de la otra cámara del mismo modo), el origen de la otra cámara se encontrará en T . Según lo anterior, la ecuación del plano epipolar a través de P puede escribirse como la condición coplanar de los vectores:

$$(P_l - T)^T (TxP_l) = 0$$

Utilizando la relación que liga a los vectores P_i y P_d se obtiene:

$$(R^T P_r)^T (TxP_l) = 0 \quad (2.9)$$

Si se tiene en cuenta que el producto vectorial de dos vectores se puede escribir como la multiplicación de una matriz antisimétrica por un vector, se tiene la siguiente expresión:

$$TxP_l = SP_l$$

Donde la matriz S se puede definir utilizando las componentes de T del siguiente modo:

$$S = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

Entonces se obtiene:

$$\begin{aligned} P_r^T E P_l &= 0 \\ E &= RS \end{aligned} \tag{2.10}$$

Por construcción, E siempre tiene rango igual a dos. Por otro lado, la matriz E se denomina matriz esencial y establece una unión natural entre la restricción epipolar y los parámetros extrínsecos del sistema estéreo.

Si ahora se divide entre $Z_l Z_r$ se obtiene:

$$p_r^T E p_l = 0 \tag{2.11}$$

Si ahora se define el vector $u_r = E p_l$ entonces se puede interpretar como el vector director de la recta en que se proyecta, sobre la imagen derecha, el rayo definido por $O_l P$. Así pues, se establece una aplicación entre los puntos de una imagen y las rectas epipolares de la otra.

Si la expresión anterior se reescribe como:

$$p_r^T u_r = 0$$

Se puede interpretar como el punto p_r que pertenece a la línea u_r .

Por lo tanto, u_r es la línea epipolar en la imagen derecha correspondiente a p_l en la imagen izquierda. Del mismo modo, se puede definir $u_l = E^T p_r$ para obtener la línea epipolar en la otra imagen.

Hasta ahora se ha trabajado en coordenadas de los sistemas de referencia asociados a las cámaras, sin embargo, cuando se miden los puntos proyección, las medidas se obtienen en términos de píxeles, por tanto para poder sacar el máximo partido de la

matriz esencial, será necesario conocer la transformación desde coordenadas de la cámara a coordenadas de píxeles, para ello se definirá el concepto de la *matriz fundamental*.

2.2.1.2 Matriz fundamental

La aplicación entre puntos y líneas puede obtenerse a partir de puntos correspondientes, sin necesidad de información a priori sobre el sistema estéreo.

Sean K_l y K_r las matrices de los parámetros intrínsecos (aquellos que especifican las características propias de la cámara tales como la distancia focal, distorsión radial, tamaño efectivo del píxel, etc.) de la cámara izquierda y derecha respectivamente. Si se denota por \bar{p}_l y \bar{p}_r a los puntos en coordenadas píxel correspondientes a p_l y p_r respectivamente, se tiene:

$$p_l = K_l^{-1} \bar{p}_l \quad p_r = K_r^{-1} \bar{p}_r$$

Sustituyendo en la ecuación de la matriz esencial se obtiene:

$$\bar{p}_l^T F \bar{p}_r = 0 \quad (2.12)$$

Donde:

$$F = K_r^{-T} E K_l^{-1} = K_r^{-T} R S K_l^{-1}$$

F se denomina la matriz fundamental. Al igual que con $E p_l$, $F \bar{p}_l$ puede ser interpretado como el vector de la recta epipolar proyectiva correspondiente al punto \bar{p}_l , es decir $\bar{u}_r = F \bar{p}_l$, del mismo modo se puede obtener $\bar{u}_l = F^T \bar{p}_r$.

La mayor diferencia entre las ecuaciones, en términos de la matriz esencial y la matriz fundamental se da, en que la matriz esencial está definida en forma de vectores en los sistemas de referencia de las cámaras, mientras que la matriz fundamental contiene vectores en coordenadas píxeles de los planos de proyección. Consecuentemente, si se estima la matriz fundamental a partir de puntos en correspondencia en coordenadas

píxel, se puede reconstruir la geometría epipolar sin absolutamente ninguna información sobre los parámetros intrínsecos o extrínsecos.

Todo lo anterior indica, que es posible establecer la correspondencia entre los puntos de una imagen y sus correspondientes líneas epipolares, sin ningún conocimiento a priori de los parámetros del sistema estéreo.

La matriz fundamental, al igual que la matriz esencial, tampoco es una matriz de rango completo, siendo de rango igual a dos. Por otro lado, la matriz fundamental codifica información sobre los parámetros tanto intrínsecos como extrínsecos.

Una consecuencia muy importante de las anteriores propiedades, es que si se conoce la calibración de un sistema estéreo o de una cámara, (parámetros intrínsecos y parámetros extrínsecos) se conoce entonces toda la información necesaria para obtener la geometría epipolar y la aplicación que liga puntos de una imagen con sus correspondientes rectas epipolares.

Para la obtención de los epipolos a partir de la matriz F se procede de la siguiente forma:

Sea \bar{e}_l , el epipolo del plano izquierdo. Dado que es un punto por el que pasan todas las rectas epipolares asociadas a los puntos de la imagen derecha, se verifica que $\bar{p}_r^T F \bar{e}_l = 0$ para todo \bar{p}_r . Pero dado que F no es idénticamente nula, esto tan solo será posible si se verifica que $F \bar{e}_l = 0$. Dado que F es de rango igual a dos, el epipolo \bar{e}_l se puede calcular como el autovector asociado al autovalor nulo de la matriz F . Haciendo un razonamiento similar, se puede demostrar que el epipolo de la imagen derecha es el autovector asociado al autovalor nulo de la matriz F^T , $F^T \bar{e}_d = 0$

Las consideraciones realizadas para la matriz E son similares a las realizadas para la matriz F .

En las siguientes figuras se muestran las líneas epipolares y el epipolo de las imágenes en distintas posiciones de la cámara respecto del objeto. En el caso de la Figura 2.8 se muestran las líneas epipolares para el caso general de dos cámaras no paralelas.

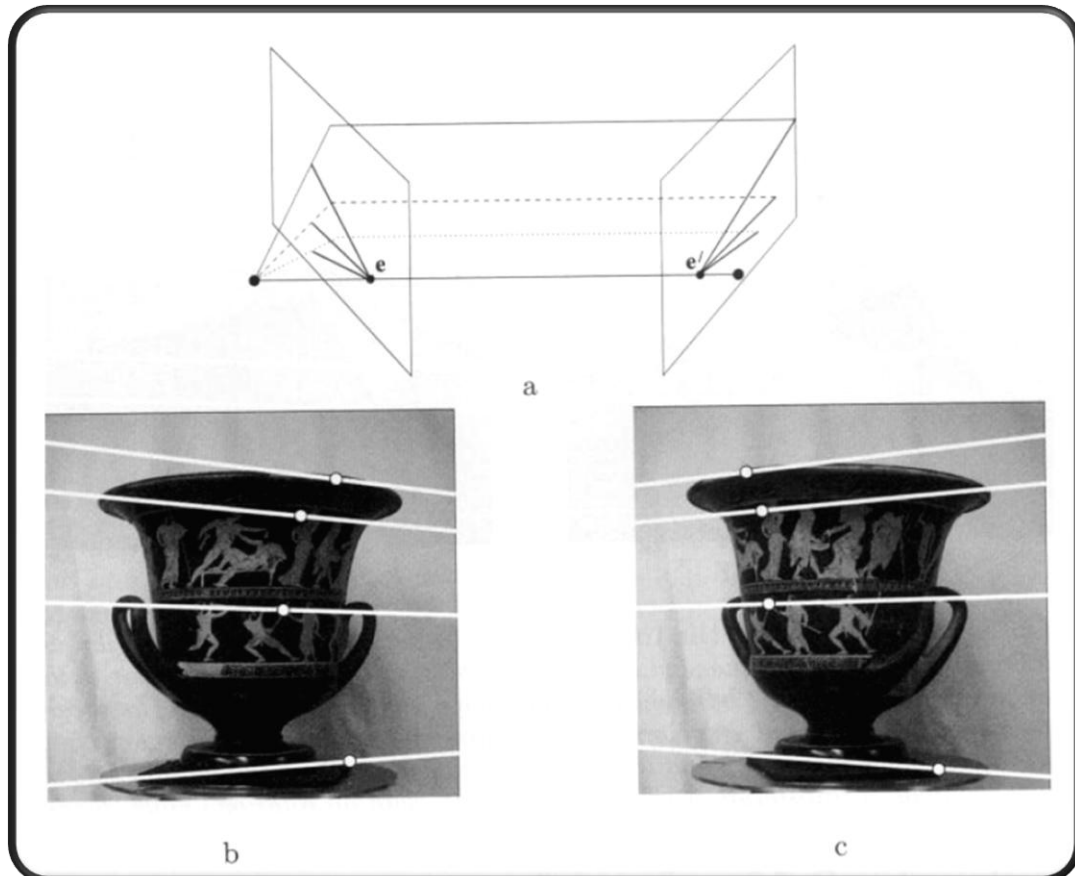


Figura 2.8 Localización de las líneas epipolares para cámaras no paralelas. Situación de las cámaras (a). Imágenes tomadas por ambas cámaras así como las líneas epipolares conjugadas (b) (c).

En la Figura 2.9 se muestran dos planos prácticamente paralelos de la misma escena. Obsérvese que las líneas epipolares son paralelas, lo que indica que los epipolos están en el infinito.

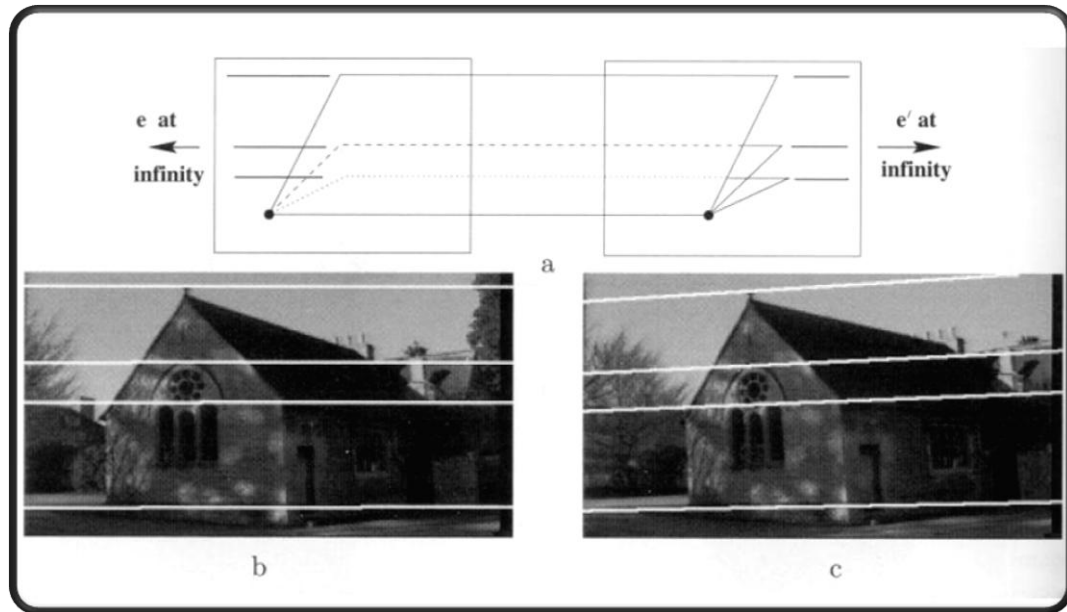


Figura 2.9 Localización de las líneas epipolares para cámaras paralelas. Situación de las cámaras (a). Imágenes tomadas por ambas cámaras así como las líneas epipolares conjugadas (b) (c).

2.2.1.3 Cálculo de la matriz fundamental o la matriz esencial

A continuación se citan varios métodos de cálculo para estas matrices, en función de los parámetros que se conocen.

- *Método de los ocho puntos:* Se parte de la hipótesis de que se conocen n pares de puntos en correspondencia entre las dos imágenes.
- *Triangulación:* Se supone que se conocen los parámetros intrínsecos y extrínsecos de las dos cámaras.
- *Estimación de posición y orientación:* Se supone que sólo son conocidos los parámetros intrínsecos de las dos cámaras.

2.2.2 Calibración

Hasta ahora, se ha supuesto el caso ideal de cámaras iguales y paralelas, por lo que la búsqueda de un punto se reducía a líneas horizontales en la otra imagen. Para el caso general de dos cámaras, únicamente conociendo un punto en una de las imágenes, su correspondiente en la otra imagen se encontrará como ya se explicó, en la línea

epipolar. En la simplificación realizada hasta ahora en la que se reducía la búsqueda a un problema de 1D en lugar de 2D, se ha basado en una serie de premisas nada fáciles de cumplir en la práctica [7], tales como: que ambas cámaras serán exactamente iguales y que los ejes ópticos sean paralelos. Esto puede verse en la Figura 2.10. Para construir el sistema estéreo se han colocado dos cámaras con ópticas del mismo modelo, y se ha procurado que la estructura mecánica asegure que estén paralelas. Puede comprobarse, siguiendo las líneas blancas horizontales, que no lo son totalmente. Aunque pueda pensarse que las diferencias son muy pequeñas, en la Figura 2.10 (c) se tiene el resultado de obtener el mapa de disparidades, en el que se observa que los resultados son erróneos. Hay que hacer notar que estos errores han sido debidos a que hemos buscado los puntos correspondientes entre las dos imágenes en las líneas epipolares que se suponían serían las líneas horizontales.

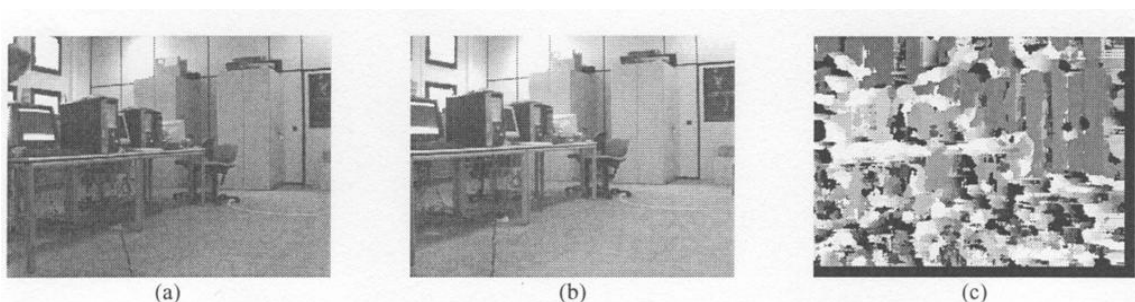


Figura 2.10 Sistema estéreo no calibrado. Imágenes izquierda y derecha originales (a) (b). Mapa de disparidades entre ambas imágenes (c).

En el caso general hubiese pasado lo mismo, porque el error de fondo está en las tolerancias en la fabricación de las cámaras, las ópticas y la base mecánica. Por lo tanto, el problema de la búsqueda de puntos correspondientes sigue siendo 2D, a menos que se realice un preprocesamiento de las imágenes estéreo.

El proceso que se utilizará para solventar este problema es conocido por el nombre de *rectificación* (Figura 2.11), que es la transformación de una pareja de sistemas ópticos en otros dos que sean colineales y donde las líneas epipolares sean paralelas a unos de los ejes de las imágenes. Se puede pensar en este proceso como si se hubiera rotado a las cámaras originales alrededor de sus centros ópticos. En los siguientes apartados se verá cómo puede realizarse esto en la práctica y los pasos a seguir.

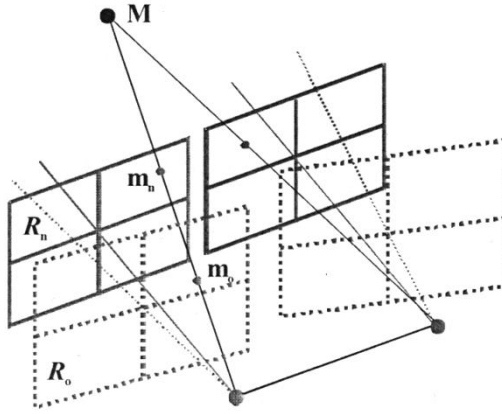


Figura 2.11 Rectificación.

2.2.2.1 Modelo de cámara en coordenadas homogéneas

Siguiendo la notación propuesta por *Ayache* [8], el modelo de la cámara (Figura 2.12) consiste en el centro óptico, c , el plano de la imagen (también denominado plano de la retina), R . Un punto 3D, $\mathbf{M} = (x, y, z)^T$ en coordenadas del mundo, se proyecta en la imagen en un punto 2D de coordenadas $\mathbf{m} = (u, v)^T$, donde \mathbf{m} es la intersección del plano R con la línea que contiene a \mathbf{M} y a c . Esto mismo, expresado en coordenadas homogéneas es:

$$\tilde{\mathbf{m}} = (U, V, S)^T$$

con:

$$u = \frac{U}{S} \quad v = \frac{V}{S} \quad \text{si } S \neq 0$$

Los puntos para los que S vale cero pertenecen al plano focal F . De esta forma, utilizando este sistema de coordenadas, estos puntos ya no son una excepción al valer infinito, sino que corresponden a aquellos puntos en los que S vale cero.

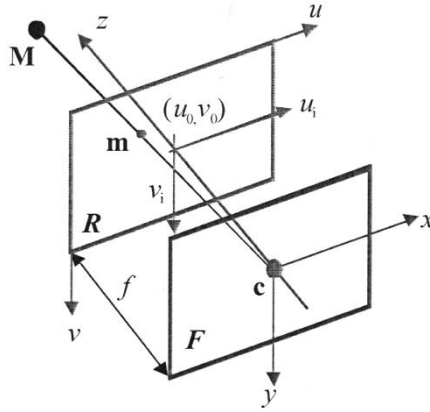


Figura 2.12 Modelo de cámara en coordenadas homogéneas.

De forma análoga:

$$\tilde{M} = (x, y, z, 1)^T$$

y la matriz de proyección de perspectiva que relaciona ambos puntos es:

$$\tilde{m} = \tilde{P}\tilde{M}$$

Por otra parte, la matriz de proyección de perspectiva se puede descomponer en el producto de otras dos matrices:

$$\tilde{P} = AG$$

Donde **A** representa los parámetros intrínsecos de la cámara y **G** los extrínsecos.

Para los parámetros intrínsecos se tiene:

$$A = \begin{pmatrix} f_u & 0 & u_0 \\ 0 & f_v & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$

Siendo (u_0, v_0) las coordenadas del corte del eje óptico con el plano de la imagen (el punto principal), respecto a la esquina superior izquierda en píxeles.

Además, por construcción se pueden tener diferencias entre ambos ejes de la lente, por lo que se debe conocer el paso de milímetros a píxeles de forma independiente (k_u, k_v) y la distancia focal de las cámaras f , con ello se obtiene:

$$\begin{aligned} f_u &= f \cdot k_u \\ f_v &= f \cdot k_v \end{aligned} \quad (2.13)$$

La matriz \mathbf{G} está compuesta por una matriz de rotación \mathbf{R} , de dimensiones 3x3, y un vector de traslación \mathbf{t} , que representan la posición y orientación respectivamente de la cámara respecto al sistema de coordenadas del mundo.

$$\mathbf{G} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \end{pmatrix}$$

También se puede definir:

$$\tilde{\mathbf{P}} = (\mathbf{P} \mid \tilde{\mathbf{p}})$$

Donde $\tilde{\mathbf{p}}$ es la última columna de la matriz $\tilde{\mathbf{P}}$.

Obviando las demostraciones que no proceden, se obtienen las expresiones:

$$\begin{aligned} \mathbf{c} &= -\mathbf{P}^{-1} \tilde{\mathbf{p}} \\ \tilde{\mathbf{p}} &= -\mathbf{P} \mathbf{c} \end{aligned} \quad (2.14)$$

Las coordenadas del centro óptico \mathbf{c} se utilizarán más tarde para la rectificación.

Y entonces:

$$\tilde{\mathbf{P}} = (\mathbf{P} \mid -\mathbf{P} \mathbf{c}) \quad (2.15)$$

Debido a que un rayo óptico asociado con un punto \mathbf{m} en la imagen es la línea que une ese punto con el centro óptico, se deriva que:

$$\begin{aligned}\tilde{m} &= P(M - c) \\ M &= c + \lambda P^{-1}\tilde{m}\end{aligned}\tag{2.16}$$

Con esta expresión y un par de imágenes estereo se podrá obtener la distancia a la que se encuentra el objeto. Como se puede observar, si el valor de su coordenada homogénea S es cero, el epipolo estará en el infinito y esto quiere decir que la proyección del centro óptico está en el plano focal de la otra cámara (Figura 2.13) y las líneas epipolares serán paralelas. Si además la línea que une ambos centros ópticos es paralela a uno de los ejes de la imagen, éstas serán horizontales o verticales.

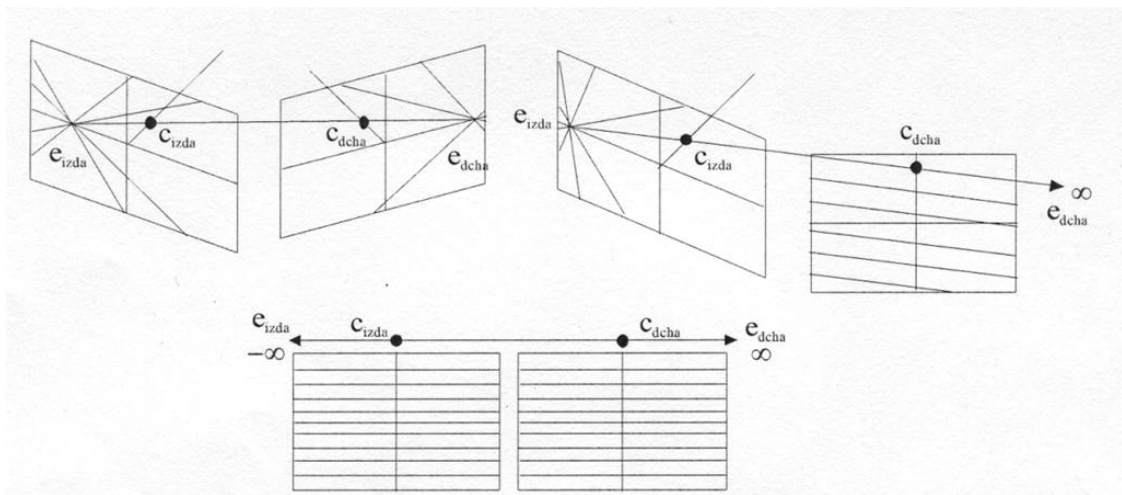


Figura 2.13 Posición de las líneas epipolares para distintos casos.

Existen diversos métodos para realizar la calibración de las cámaras, los cuales se explicarán a continuación:

- Calibración pasiva.
- Calibración activa.
- Calibración de la cámara por sí misma.

2.2.2.2 Calibración pasiva

Es la clásica técnica de calibración de la cámara. Los parámetros de ésta (extrínsecos e intrínsecos) se asumen como fijos, por lo que sólo es necesario que se realice una vez.

El procedimiento básico es:

1. Determinar con precisión un conjunto de puntos tridimensionales.
2. Determinar sus correspondientes proyecciones en la imagen.
3. Obtener los parámetros que mejor resuelven la correspondencia entre unos y otros.

Para todo ello es necesario generar o adquirir una *plantilla de calibración* que incluya marcas para las que fácilmente se pueda conseguir las correspondencias entre puntos de la imagen y puntos del mundo real, de modo manual o automático (Figura 2.14).



Figura 2.14 Patrón de calibración.

Para llevar a cabo este proceso, se suele utilizar el *Camera Calibration Toolbox* para MATLAB® [9], que se está convirtiendo en el software estándar de calibración. Principalmente, este Toolbox implementa el método desarrollado por *Zhang* [10] [11]. Requiere de un patrón de calibración plano, que mediante unos puntos definidos por el usuario, el algoritmo obtiene los parámetros deseados. Como se puede observar en la Tabla 2.1, estos valores serían un ejemplo de la salida de dicho algoritmo:

Cámara	k_1	k_2	p_1	p_2	u_0	v_0	f_x	f_y
Izquierda	-0.24386	0.19161	0.00254	-0.00761	308.49	237.40	815.03	806.27
Derecha	-0.25275	0.32041	0.00470	-0.00905	301.74	226.90	815.40	807.86

$$R = \begin{pmatrix} 1.0000 & -0.0047 & 0.0055 \\ 0.0047 & 1.0000 & -0.0025 \\ -0.0055 & 0.0025 & 1.0000 \end{pmatrix} \quad t = (-147.86 \quad 0.96 \quad 11.3)^T$$

Tabla 2.1 Valores de calibración

Donde k_1 y k_2 son los coeficientes de distorsión radial, p_1 y p_2 los coeficientes de distorsión tangencial, u_0 y v_0 las coordenadas del punto principal respecto a la esquina superior izquierda (en píxeles), f_x y f_y las distancias focales para cada eje.

R es la matriz de rotación y t el vector de traslación según el sistema de referencia que se observa en la Figura 2.15.

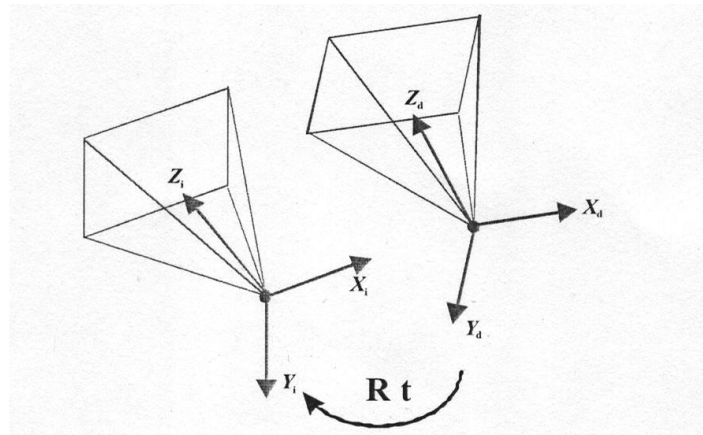


Figura 2.15 Sistemas de coordenadas.

2.2.2.3 Calibración activa

En la calibración activa de la cámara, los parámetros intrínsecos de la cámara se suponen variables, mientras que los parámetros extrínsecos son fijos [12]. El proceso de calibración tiene que considerar el rango completo de los parámetros variables, como por ejemplo, el uso del zoom y el autoenfoco, ya que modifican los parámetros de calibración.

El objetivo de esta calibración es desarrollar técnicas robustas y en tiempo real para mantener una precisión en la determinación de los parámetros extrínsecos como intrínsecos. Cabe destacar que el algoritmo de calibración se debe ejecutar continuamente para poder conocer los cambios que se producen en la cámara.

2.2.2.4 Calibración de la cámara por sí misma

En realidad, se podría considerar como un método de calibración activa. El propósito de este tipo de calibración es poder recuperar los parámetros de la cámara en distintas configuraciones del sistema de visión. Este proceso se desarrolla en dos pasos:

1. Estimar los parámetros del sistema en su posición inicial.
2. Actualizar los parámetros intrínsecos basados en el factor del zoom de las lentes de la cámara.

2.2.2.5 Rectificación

Una vez que se conocen los parámetros intrínsecos y extrínsecos (después de calibrar el sistema) entre el sistema de coordenadas del mundo y el de cada cámara, se puede proceder a la aplicación de un *método de rectificación*, mediante el cual se obtendrán unas nuevas matrices de perspectiva que cumplirán las siguientes condiciones:

- Debe preservar los centros ópticos.
- Los planos focales deben de ser coplanares, por lo que contienen la recta que une los centros ópticos. Con esta condición los epipolos están en el infinito y las líneas epipolares son paralelas.
- Además de paralelas, se quiere que las líneas epipolares sean horizontales, para ellos la recta anterior debe de ser paralela al eje horizontal de ambas cámaras.
- Por último, los puntos conjugados, los de ambas imágenes que corresponden al mismo punto del mundo deben de tener la misma coordenada vertical, por lo que las cámaras poseen los mismos parámetros intrínsecos.

Una vez obtenidas las nuevas matrices de perspectiva, se podrá obtener la transformación de rectificación que aplicada a una imagen consiga todos los requisitos nombrados anteriormente.

Para ver la importancia de la rectificación y los resultados se pueden observar la Figura 2.16. Las dos primeras imágenes Figura 2.16 (a) y Figura 2.16 (b) no están rectificadas y se aprecia que no son totalmente paralelas. Aunque pueda pensarse que las diferencias son muy pequeñas, en la Figura 2.16 (e) se presenta el resultado de obtener el mapa de disparidades en el que se observa que los resultados son erróneos. El contraste con los resultados mostrados en la Figura 2.16 (f), a partir de las imágenes rectificadas Figura 2.16 (c) y Figura 2.16 (d) es notable.

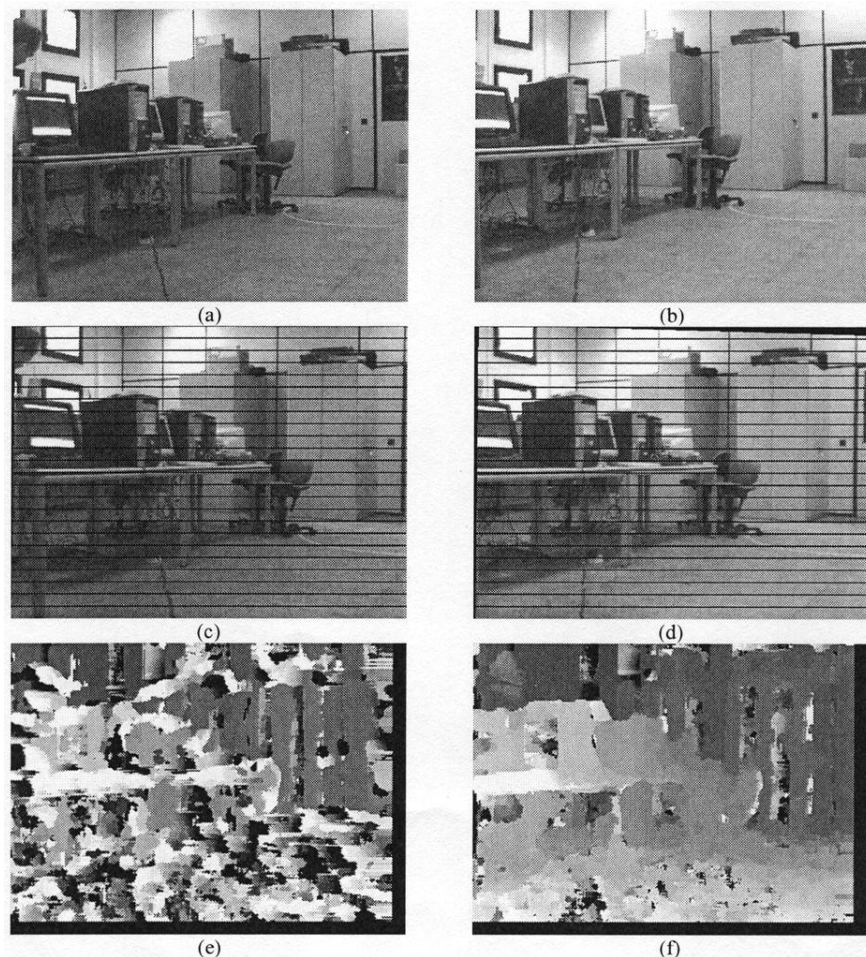


Figura 2.16 Resultados. Imágenes izquierda y derecha sin rectificar (a) (b). Imágenes izquierda y derecha rectificadas (c) (d). Mapa de disparidades entre a y b (e). Mapa de disparidades entre c y d (f).

2.2.3 Correspondencia

Una vez que se ha realizado la calibración del sistema, se llega al paso más importante del problema de visión estéreo, que es el cálculo de la correspondencia de un punto de una imagen con la tomada por la otra cámara.

Una primera decisión es qué tipo de objetos se van a buscar en la imagen. Así se tienen algoritmos que buscan puntos característicos de la imagen (bordes, esquinas) o los que tratan de emparejar áreas. Los primeros son algoritmos rápidos, pero que a cambio obtienen mapas de disparidad no densos. Los segundos son mucho más lentos, pero los mapas son densos. Estos últimos fallan cuando hay poca textura, por lo que las áreas son muy parecidas, o con la presencia de oclusiones, ya que la disparidad varía rápidamente dentro de la ventana de correlación. Es este segundo grupo el que se va a seguir revisando y el que se centra este proyecto.

Uno de los principales problemas que se plantean es la velocidad de procesamiento. Para realizar la correlación y que el algoritmo no sea demasiado lento, es necesario tener en cuenta una serie de restricciones:

- **Restricción fotométrica:** Las intensidades entre las dos zonas deben ser iguales o al menos parecidas. Para ello se supone que los objetos serán superficies lambertianas (reflejan la misma luz en todas direcciones, por lo que se verán iguales en las dos cámaras). Esta condición fallará cuando no sean así las superficies (presencia de especularidades) y se verá condicionada por la diferente ganancia de las cámaras o la distinta apertura de los diafragmas. En lugar de comparar píxel a píxel, se toma una ventana cuyas dimensiones pueden crear problemas adicionales [13].
- **Restricción geométrica:** Se reduce la búsqueda a la línea epipolar. Por lo tanto se reduce el espacio de búsqueda de 2D a 1D.
- **Restricción de la disparidad:** Se delimita la búsqueda a un rango de disparidades entre un valor mínimo y máximo.

2.2.3.1 Correlación

Para realizar la correlación, lo que se hace, en términos generales, es coger una ventana alrededor de un píxel central y realizar la búsqueda de esta ventana en la imagen tomada por la otra cámara. El tamaño de la ventana es uno de los factores más importantes por lo que se estudiará en apartados posteriores con mayor detalle.

Durante este proyecto se contemplaron únicamente los *métodos paramétricos* para el cálculo de dicha correspondencia. A continuación se enumeran varios métodos en los cuales, los valores I_1 e I_2 hacen referencia al nivel de gris de cada cámara, y los sumatorios indican que se recorre la ventana de búsqueda.

- Suma de diferencias absolutas.

$$SAD = \sum_{i,j} |I_1(i, j) - I_2(x+i, y+j)| \quad (2.17)$$

- Suma de diferencias cuadradas.

$$SSD = \sum_{i,j} I_1(i, j) - I_2(x+i, y+j)^2 \quad (2.18)$$

- Suma normalizada de diferencias cuadradas.

$$NSSD = \frac{\sum_{i,j} I_1(i, j) - I_2(x+i, y+j)^2}{\sqrt{\sum_{i,j} I_1(i, j)^2 \sum_{i,j} I_2(x+i, y+j)^2}} \quad (2.19)$$

- Correlación normalizada.

$$NCC = \frac{\sum_{i,j} I_1(i, j) \cdot I_2(x+i, y+j)}{\sqrt{\sum_{i,j} I_1(i, j)^2 \sum_{i,j} I_2(x+i, y+j)^2}} \quad (2.20)$$

Estos métodos tienen el problema, para su puesta en la práctica, de la diferencia de ganancias entre las cámaras (con lo que la restricción fotométrica presenta problemas). Por ejemplo, si una cámara captara la imagen con la mitad de iluminación que la otra, el resultado que se obtendrá al restar ambos valores no será un indicador *fiabile* de correlación. Por lo tanto para evitar este fenómeno se plantean tres opciones, donde \bar{I}_1 e \bar{I}_2 son el nivel medio de gris de la ventana de búsqueda.

- Suma de diferencias absolutas con media cero.

$$ZSAD = \sum_{i,j} |(I_1(i, j) - \bar{I}_1) - (I_2(x+i, y+j) - \bar{I}_2)| \quad (2.21)$$

- Suma de diferencias cuadradas con media cero.

$$ZSSD = \sum_{i,j} ((I_1(i, j) - \bar{I}_1) - (I_2(x+i, y+j) - \bar{I}_2))^2 \quad (2.22)$$

- Preprocesar ambas imágenes mediante una Laplaciana de Gaussiana y posterior aplicación del algoritmo SSD ó SSA (los de menor coste computacional).

Como se justificó anteriormente, el criterio que se usará para las elecciones será el coste computacional. Es por ello por lo que la solución primera se descarta, ya que para su uso será necesario calcular los valores medios de cada ventana (para cada valor de disparidad) y para cada píxel de ambas imágenes. Por lo tanto, la opción que se utilizará para su implementación será la segunda, ya que no requiere dicho cálculo de valores medios, debido a que la aplicación de una Laplaciana de una Gaussiana a una imagen deja un valor medio de gris constante, por lo que se disminuye el efecto de la iluminación, ganancia de las cámaras, y se resaltan los detalles de la imagen.

2.2.3.2 Obtención del máximo y de la disparidad

Una vez se ha calculado la correspondencia para cada píxel con todos los de la otra imagen, se procede al cálculo del punto que mejor se asemeja de los candidatos. Para ello, si por ejemplo se ha utilizado el método SSD (suma de diferencias cuadradas) o el SAD (suma de diferencias absolutas) se buscará el punto de menor valor del mismo.

Este método tiene una precisión de píxel, ya que se está considerando implícitamente que la función que relaciona la correspondencia con la disparidad es lineal, y por lo tanto su máximo o mínimo siempre estará en uno de los píxeles de la imagen en la que se está buscando, como se puede ver en la Figura 2.17.

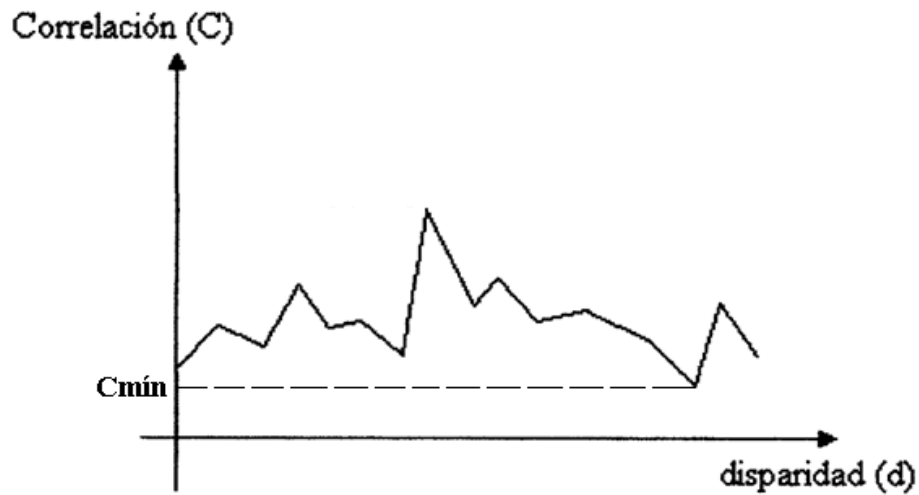


Figura 2.17 Disparidad con precisión de píxel.

Otros autores [14] [15] proponen obtener la parábola definida por el mínimo de la correlación y sus dos vecinos, obteniéndose así la disparidad con precisión subpíxel como se puede ver en la Figura 2.18.

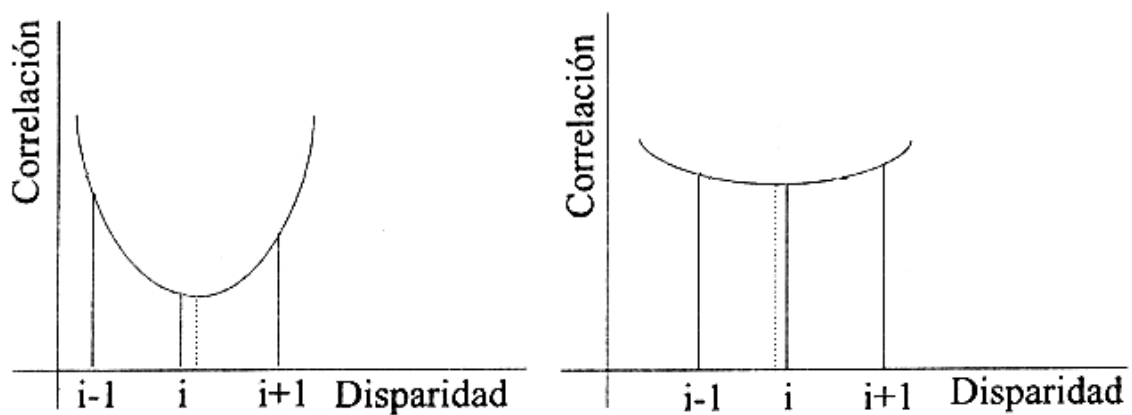


Figura 2.18 Disparidad con precisión subpíxel.

Se puede definir la parábola mediante la siguiente fórmula:

$$C = a(i - i_0)^2 + C_0$$

Siendo C el valor de la correlación, i_0 la disparidad que da la correlación máxima C_0 con precisión subpíxel y que se quiere encontrar. Entonces obviando las demostraciones se obtienen:

$$\begin{aligned} a &= \frac{2C_2 - (C_1 + C_3)}{2} \\ i_0 &= i + \frac{C_3 - C_1}{2(2C_2 - (C_1 + C_3))} \\ C_0 &= \frac{C_1 + 4C_2 - C_3}{4} \end{aligned} \tag{2.23}$$

Siendo C_1 , C_2 , C_3 los valores de la correlación para los puntos $i-1$, i , $i+1$ respectivamente.

Por otra parte i_0 es la disparidad que da la correlación máxima C_0 con precisión subpíxel que se quiere encontrar.

2.2.4 Filtrado de la imagen de disparidades

Un mapa de disparidades es una imagen, donde la intensidad de tono en cada píxel indica la distancia horizontal a la que se encuentra el píxel correspondiente de una imagen en la otra. A cada píxel (x, y) en el mapa de disparidad se le asigna el valor de disparidad d , donde se encontró el mejor valor de correspondencia expresado en unidades de distancia (píxeles). De esta manera dicho mapa contendrá el valor de disparidad para cada píxel, en la cual se encontró la mejor correlación con la ventana de píxeles buscada.

A la hora de representar dichos valores en la imagen, se suelen normalizar de $[0 \ 255]$ para ayudar a percibir mejor las diferencias entre los distintos niveles de gris.

Una vez se ha llegado a este punto, se puede pasar a la siguiente etapa, en la que se intenta, mediante técnicas de filtrado, conseguir la restricción de unicidad.

Con todo ello, se pretende mejorar los resultados obtenidos despreciando las disparidades que dan resultados erróneos por múltiples causas, como se puede observar en la Figura 2.19, para ello existen numerosos métodos, entre ellos:

- Eliminación de zonas con poca textura. Esto es debido a que van a dar múltiples resultados y la certeza de la medida es pequeña.
- Valor mínimo de la correlación que se acepta como válido.
- Patrones repetitivos. Analizando los máximos de la correlación se podrán eliminar oclusiones y patrones repetitivos.
- Comprobación *cross-checking* [16] [17]. Este método sirve para desechar malas medidas debidas a oclusiones, para ello se compara el mapa de disparidad izquierdo con el derecho (y a viceversa), y en función de las diferencias encontradas se establecerá o desechará un valor de disparidad.
- Postprocesamiento. Se aplica el filtro de la mediana al mapa de disparidades. Con ello se consigue la eliminación de disparidades aisladas, debidas a falsas medidas por oclusiones.

2.2.5 Problemas computacionales de la visión estéreo

Una vez que se han enumerado las diversas posibilidades para el cálculo de la correspondencia, debe destacarse el **elevado coste computacional** para su cálculo en tiempo real (aproximadamente 30 FPS). Hay que destacar que para el cálculo de la correlación, los métodos anteriores se aplican para cada píxel de la imagen, y para cada valor de disparidad. Es por este motivo por el que hasta hace poco tiempo se había considerado inviable en términos de hardware necesario, para aplicaciones pequeñas.

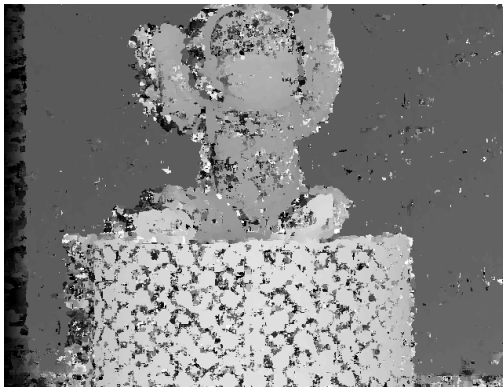
El motivo de este proyecto es calcular dichas correspondencias (así como el máximo de las mismas) haciendo uso de la nueva tecnología NVIDIA® CUDA™, que permite mediante la utilización de la GPU, paralelizar los cálculos haciéndose uso de los numerosos núcleos que poseen las tarjetas gráficas en la actualidad. Con todo ello será posible con un bajo coste, la realización de cálculos en unos tiempos de cómputo que hasta la fecha parecían impensables.



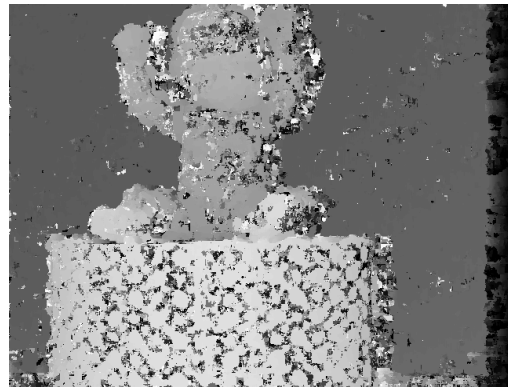
(a)



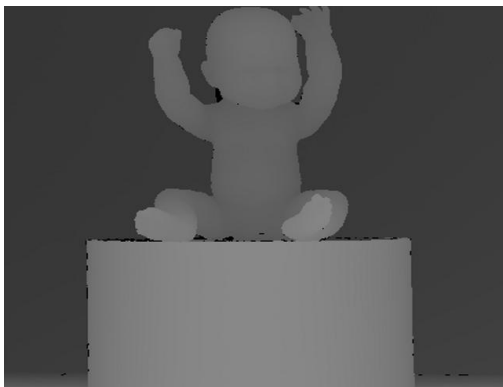
(b)



(c)



(d)



(e)



(f)

Figura 2.19 Imágenes izquierda y derecha (a) (b). Mapa de disparidades sin filtrar (c) (d). Mapa de disparidades filtrado (e) (f).

CAPÍTULO 3. INTRODUCCIÓN A NVIDIA® CUDA™

3.1 Un modelo de programación paralela y escalable

La aparición de las CPUs multinúcleo y las GPUs, también con numerosos núcleos, hace que la mayoría de los procesadores sean ahora sistemas en paralelo. Además dicho paralelismo continúa ampliándose según la famosa ley de Moore. El reto es conseguir desarrollar aplicaciones, que de manera transparente, consigan un aumento de rendimiento al aumentar el número de núcleos de un PC (en nuestro caso cambiar de tarjeta gráfica) de la aplicación, y que de un modo “invisible” al usuario aumente su grado de paralelismo.

CUDA es un modelo de programación en paralelo, así como un entorno de software diseñado para abordar este reto de un modo sencillo para los programadores familiarizados con lenguajes de programación estándar, tales como C.

Se tienen tres puntos clave: jerarquía de grupos de hilos, memoria compartida y las directivas de sincronización, las cuales están expuestas de manera sencilla como un conjunto de extensiones de C.

Mediante estos puntos se guía al programador a partir el problema en subproblemas secundarios, que se pueden resolver independientemente en paralelo, y dentro de ellos en partes más pequeñas que se podrán resolver cooperando entre hilos de ejecución paralela. Un programa compilado en CUDA puede, por lo tanto, ejecutarse en cualquier número de núcleos de procesamiento, y sólo el sistema *runtime* necesita saber el número total de ellos.

3.2 Ventajas del uso de la GPU

La elección de una tarjeta gráfica para programar en paralelo se debe principalmente a estas razones:

- Mercado insaciable en continuo desarrollo, por lo que se asegura un desarrollo seguro de esta tecnología además de una reducción drástica de precios.
- Tremendo poder de cálculo y ancho de banda comparado con las CPUs (Figura 3.1 y Figura 3.2).
- Mucho menor coste a igualdad de GFLOPS/s si se compara con CPUs convencionales.

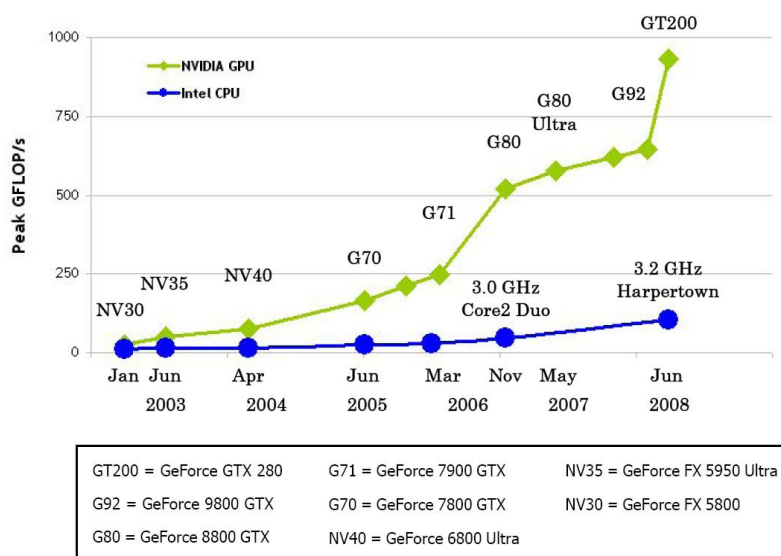


Figura 3.1 Comparación evolutiva entre los GFLOP/s para la CPU y la GPU.

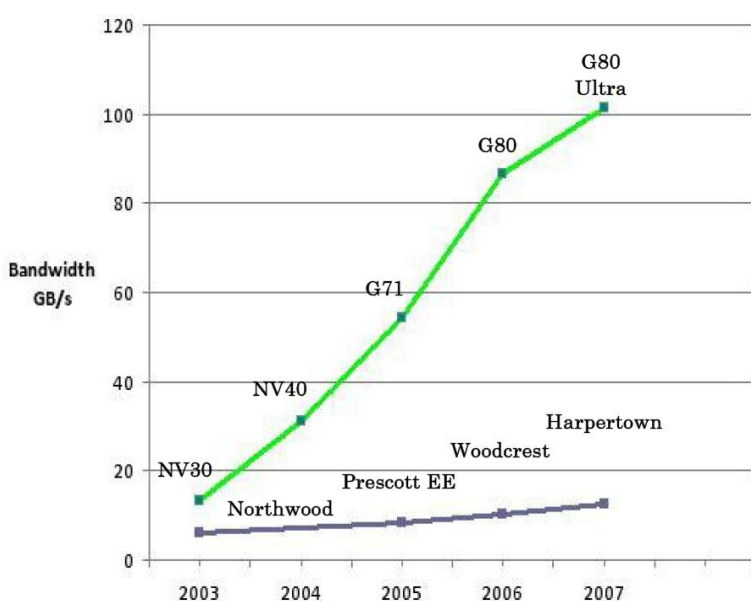


Figura 3.2 Comparación evolutiva entre el ancho de banda (GB/s) para la CPU y la GPU.

La principal razón de estas diferencias en operaciones en coma flotante se debe a que la GPU está especializada en computación intensiva y extremadamente paralela, ya que es en lo que se basa el renderizado de gráficos [18], por lo que más transistores están dedicados al procesamiento de datos como se puede ver en la Figura 3.3.

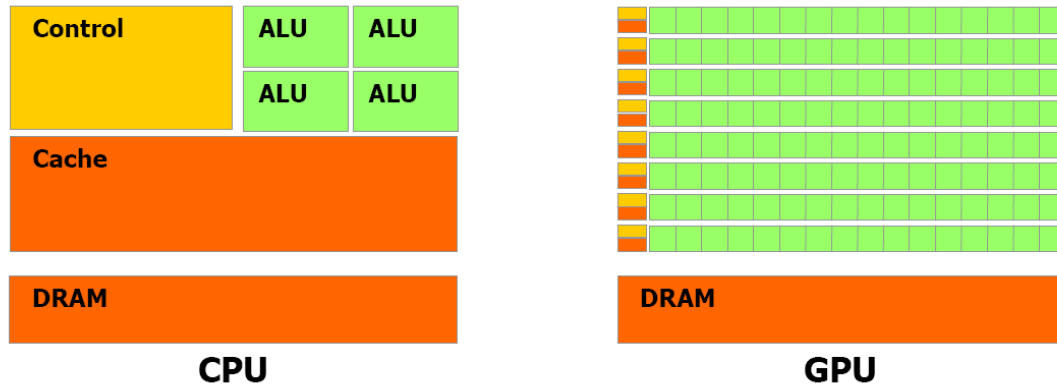


Figura 3.3 La GPU dedica más transistores al procesamiento de datos.

Más específicamente, la GPU principalmente es adecuada para direccionar problemas que se pueden expresar como operaciones de datos en paralelo (el mismo programa se ejecuta para muchos datos en paralelo). De este modo, el mismo programa se ejecuta para cada elemento y por ello requiere menor control de flujo. Además, al ejecutarse sobre muchos datos con una intensidad aritmética elevada, la latencia en el acceso a memoria se puede ocultar con cálculos en vez de con grandes cachés de datos.

La mayoría de las aplicaciones que requieren procesar grandes cantidades de conjuntos de datos pueden usar un modelo de programación *data-parallel* para acelerar los cálculos. De hecho para aplicaciones de imagen o vídeo se pueden paralelizar para obtener unas reducciones en tiempos computacionales muy elevadas.

El modelo de programación CUDA es muy adecuado para utilizar las capacidades de paralelismo que nos ofrecen las GPUs. La última generación de GPUs NVIDIA admiten

este modelo de programación y aceleran tremendamente las aplicaciones CUDA (en la Figura 3.4 se muestra una GPU compatible con dicha tecnología).



Figura 3.4 Tarjeta gráfica NVIDIA® GeForce® GTX 295 compatible con la tecnología CUDA.

3.3 Modelo de programación en CUDA

CUDA amplía el lenguaje C permitiendo al programador definir funciones llamadas *kernels* (ejecutadas en la GPU), que cuando se llaman, se ejecutan N veces en paralelo por N diferentes hilos (*threads*) y no una sola vez como las funciones típicas de C.

Un kernel se define usando la declaración `__global__` previo a las declaraciones convencionales de C (tales como `void`, `int`,...). Cuando se realice la llamada a este kernel, se definirá el número de hilos a lanzar utilizando la sintaxis `<<<...>>>`.

Cada uno de los hilos que ejecuta un kernel tiene un único *thread ID* y que es accesible dentro del kernel a través de la variable **`threadIdx`**. Para ilustrar lo anterior, el siguiente código de ejemplo suma dos vectores A y B de tamaño N y almacena el resultado en el vector C.

```
__global__ void vecAdd(float* A, float* B, float* C)

{

    int i = threadIdx.x;
```

```

    C[i] = A[i] + B[i];

}

int main()

{

    // Kernel invocation

    vecAdd<<<1, N>>>(A, B, C);

}

```

Más específicamente, se puede definir un *grid* de bloques de hilos. Cada bloque tendrá las dimensiones más adecuadas para optimizar el rendimiento global de la aplicación.

La variable que define el tamaño del bloque o del grid es del tipo *dim3*, y es un vector de tres dimensiones.

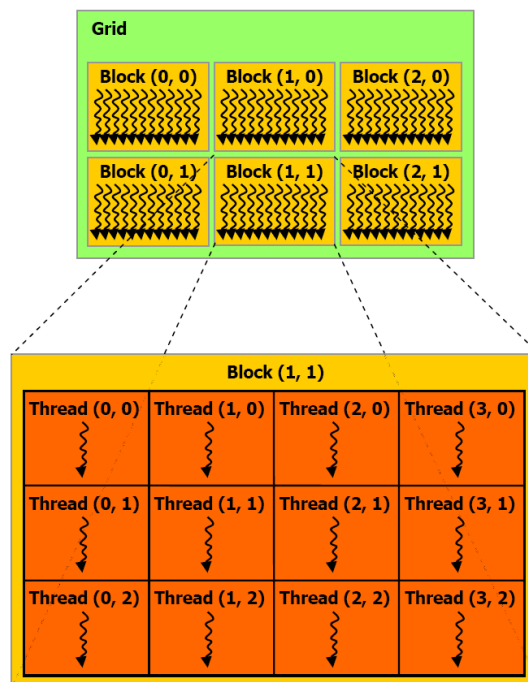


Figura 3.5 Grid de bloques de hilos.

Cada bloque del grid puede ser identificado mediante la variable tridimensional **blockIdx** (accederemos a la coordenada x por ejemplo mediante **blockIdx.x**). Del mismo modo podemos obtener las dimensiones del bloque mediante la variable

tridimensional **blockDim**. En la Figura 3.5 se puede observar un grid de bloques de hilos así como las coordenadas de cada elemento.

Un ejemplo de uso se puede ver en el siguiente código en el que se suman las matrices A y B, almacenando el resultado en la matriz C.

```
__global__ void matAdd( float A[N][N], float B[N][N],  
  
                      float C[N][N])  
  
{  
  
    int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    int j = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (i < N && j < N)  
  
        C[i][j] = A[i][j] + B[i][j];  
  
}  
  
int main()  
  
{  
  
    // Kernel invocation  
  
    dim3 dimBlock(16, 16);  
  
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,  
  
                (N + dimBlock.y - 1) / dimBlock.y);  
  
    matAdd<<<dimGrid, dimBlock>>>(A, B, C);  
  
}
```

Cada bloque es de dimensiones 16x16 hilos y se eligió arbitrariamente. El tamaño del grid se elige de forma que sea suficiente para tener un hilo para cada elemento de la matriz.

3.4 Jerarquía de memoria

Los hilos de CUDA pueden acceder a datos desde múltiples espacios de memoria durante su ejecución, como se muestra en la Figura 3.6. Cada hilo tiene su memoria privada local. Cada bloque de hilos tiene una memoria compartida visible por todos los hilos del mismo y con el mismo tiempo de vida que el bloque. Por último, todos los hilos tienen acceso a la misma memoria global.

También se tienen dos espacios de memoria de sólo lectura accesibles por todos los hilos: la memoria de tipo constante y las texturas. La memoria global, constante y las texturas están optimizadas para diferentes usos de la misma, y más adelante se justificará el uso de cada una de ellas.

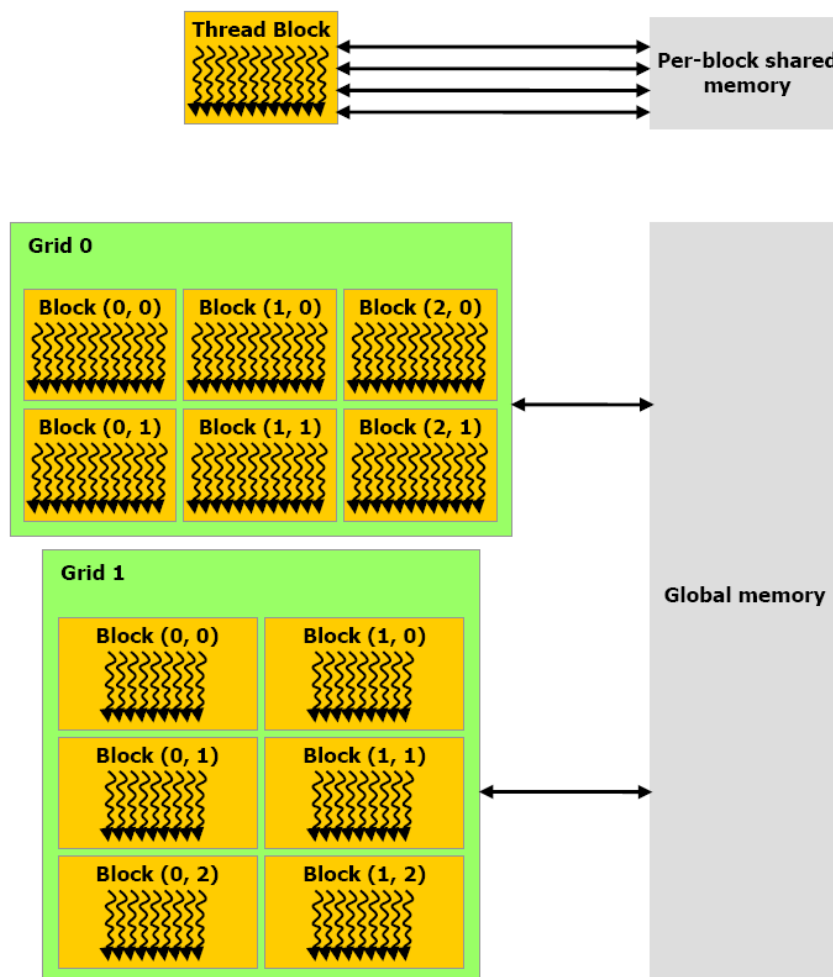


Figura 3.6 Jerarquía de memoria.

3.5 Modelo de ejecución

- Los *kernels* se lanzan en *grids*. Un *kernel* se ejecuta cada vez.
- Un bloque de hilos se ejecuta en un multiprocesador.
- Varios hilos pueden residir concurrentemente en un multiprocesador. El número está limitado por los recursos de nuestro hardware.
- Los **registros** se reparten entre todos los hilos residentes.
- La **memoria compartida** se reparte para todos los bloques de hilos residentes.

3.6 Host (CPU) y Device (GPU)

Como se ilustra en la Figura 3.7, CUDA asume que los hilos se pueden ejecutar sobre un *device* físicamente separado y que opera como coprocesador de ayuda al *host*, que está ejecutando el programa en C. Este es el caso, por ejemplo, en el que los *kernels* se ejecutan en la GPU y el resto del programa se ejecuta en la CPU.

CUDA también asume que el *host* y el *device* mantienen su propia DRAM, referida como *host memory* y *device memory*, respectivamente. Por lo tanto, un programa gestiona la memoria global, constante y las texturas de forma visible a los *kernels* a través de llamadas al CUDA *runtime*.

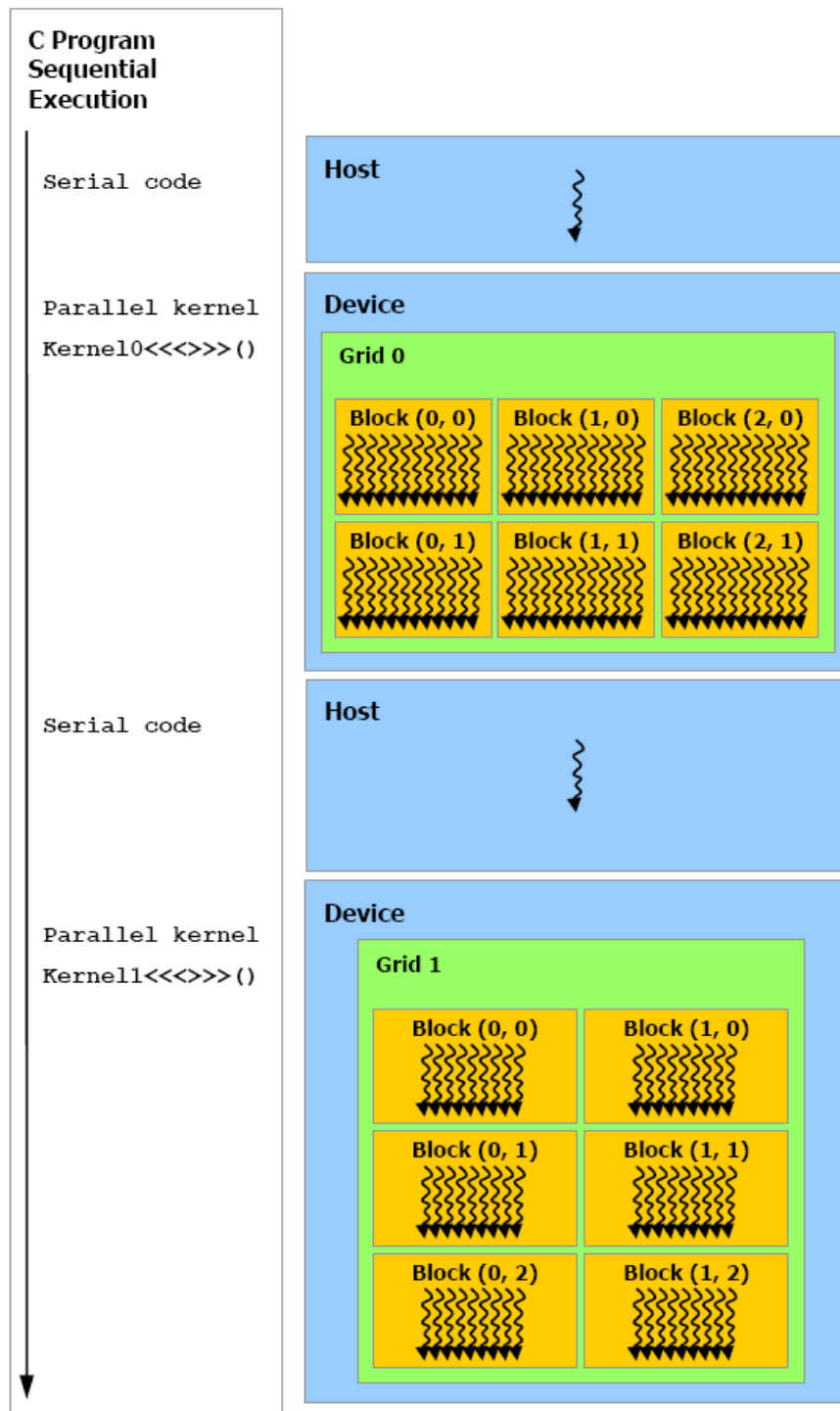


Figura 3.7 Programación heterogénea. El código se ejecuta en serie en el *device* mientras que en el *host* se ejecuta en paralelo.

3.7 Hardware GPU

Cuando un programa CUDA invoca a un kernel, los bloques del grid son enumerados y distribuidos a los multiprocesadores con capacidad disponible de ejecución. Los hilos de un bloque se ejecutan concurrentemente en un multiprocesador. Cuando terminan los bloques de hilos lanzados, nuevos bloques serán lanzados en los multiprocesadores vacantes.

Un multiprocesador consiste en ocho núcleos de Procesadores Escalares (SP), dos unidades de funciones especiales, una unidad de instrucciones multihilo y una memoria *on-chip* compartida (*shared*). El multiprocesador crea, gestiona y ejecuta hilos concurrentes. Implementa la instrucción `__syncthreads()` (hace que un hilo espere a que todos los que se están ejecutando lleguen hasta ese punto) con una simple instrucción.

Como se observa en la Figura 3.8, cada multiprocesador tiene una memoria *on-chip* de los siguientes cuatros tipos:

- Un conjunto de registros locales de 32-bit por procesador.
- Un *cache de datos paralelo o shared memory* que es compartida por todos los núcleos de procesadores escalares y que es donde la memoria compartida (*shared memory*) reside.
- Una *cache constante* de solo lectura que es compartida por todos los núcleos de procesadores escalares, y que acelera las lecturas desde la memoria constante.
- Una *cache de texturas* de sólo lectura que es compartida por todos los núcleos de procesadores escalares, y que acelera las lecturas desde la memoria para texturas.

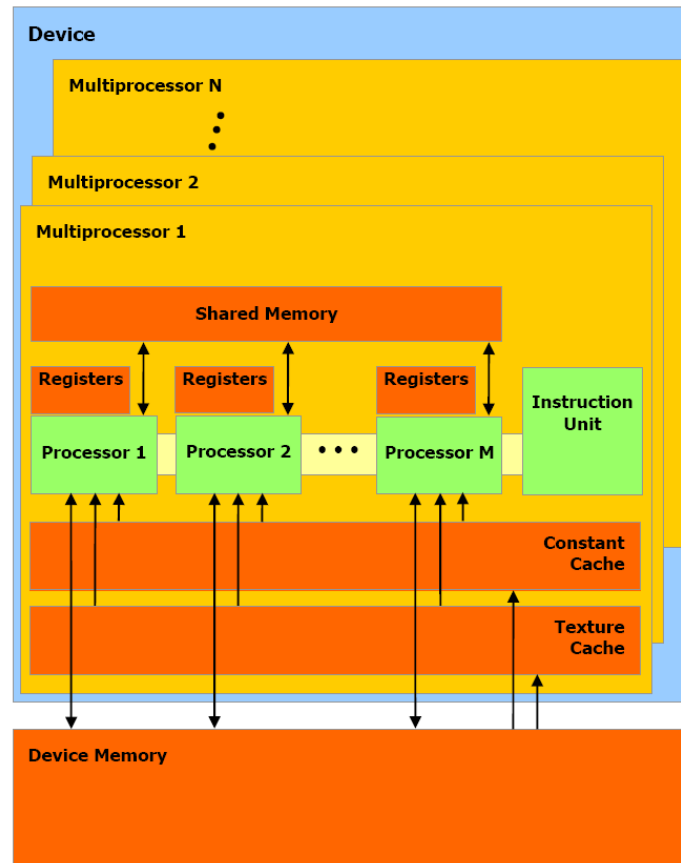


Figura 3.8 Modelo de Hardware de la GPU.

3.8 Compilar aplicaciones CUDA

CUDA tiene su propio compilador llamado **nvcc**, el cual tiene como misión generar el código objeto que se ejecutará en la GPU (denominado *cubin*), mientras que el código host (el que se ejecutará en la CPU) se compilará utilizando:

- En plataformas Linux el compilador de GNU **gcc**.
- En plataformas Windows el compilador de Microsoft Visual Studio **cl**.

La finalidad del código host (el de la CPU) es iniciar la aplicación, transfiriendo el código cubin a la GPU, reservando la memoria necesaria en el dispositivo y llevando a la GPU los datos de partida con los que se va a trabajar.

3.9 Consejos para la programación con CUDA

- Descomponer el programa en una secuencia de pasos (Grids).
- Descomponer el grid en bloques independientes y paralelos (Bloques de hilos).
- Descomponer el bloque en elementos que cooperen en paralelo (Hilos).

Para decidir de una forma eficiente el tamaño del grid y el de los bloques de hilos, es necesario conocer primero las especificaciones de nuestra GPU, las cuales se pueden consultar en la web del fabricante [19], o consultando el APÉNDICE E

Una vez conocido el *warp size* (número de hilos lanzados simultáneamente), la cantidad máxima de memoria que se puede leer y escribir por cada hilo, etc. se podrá hacer una primera estimación de un tamaño óptimo de bloque (el cual debe ser divisible por 16 para evitar problemas de gestión de memoria), sabiendo que está limitado en función de la memoria utilizada por cada hilo.

Acerca del uso de memorias, se recomienda el uso de *texturas* (cache) para mejorar el rendimiento, ya que las lecturas son mucho más rápidas y permite que sean simultáneas entre los hilos de un mismo bloque.

Es muy importante el uso de memoria compartida entre hilos, ya que es el mejor método de cooperación entre hilos y donde en realidad se hace uso del procesamiento en paralelo. Es el punto más complicado para depurar los algoritmos programados pero el más eficaz.

Otro punto de crítica importancia es el conflicto de acceso a memoria compartida. Como se ha explicado es muy importante el uso de memoria compartida, pero esto tiene unas dificultades importantes como el control de que no se accede a la misma posición de memoria en dos hilos simultáneamente. Por lo cual debe prestarse especial atención a la lectura y escritura de la memoria compartida, e intentar que si dos hilos van a acceder a una misma posición que lo hagan mediante un *semáforo* para evitar colisiones y asegurarnos de que el dato que se lee o escribe es en el momento adecuado. En las Figura 3.9 se pueden observar patrones de acceso a memoria con y sin conflictos.

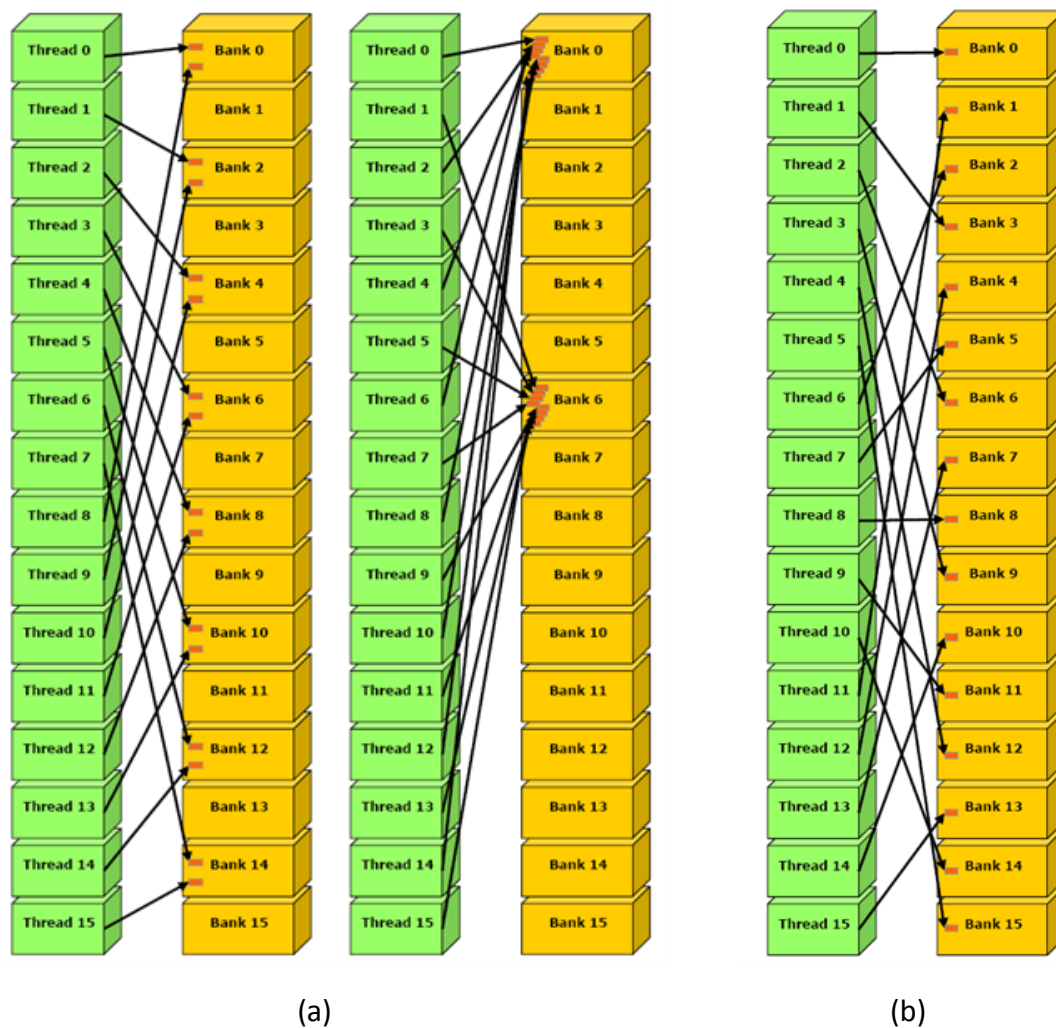


Figura 3.9 Patrón de acceso a memoria compartida con conflictos (a) y sin conflictos (b).

El último y más importante punto acerca de la programación en paralelo, es evitar la divergencia entre hilos, ya que los hilos por la arquitectura de la GPU se ejecutan tremendamente más rápido si todos siguen la misma línea del código, es decir, todos ejecutan la misma instrucción simultáneamente. Esto significa que se debe evitar el uso de condiciones en los códigos cuando sea posible, y cuando se introduzcan instrucciones que puedan hacer divergir a los hilos, se controlará su sincronismo haciendo un uso inteligente de la función `__syncthreads()`, y se prestará una especial atención especial a los bucles que puedan incrementar todavía más la desincronización de los mismos. Por lo tanto, hay que idear métodos que aprovechen el paralelismo, de

forma que casi todos los hilos realicen las mismas funciones (y cooperando todos ellos mediante la memoria compartida) y sólo unos pocos diverjan si es necesario.

3.10 Justificación de la elección de NVIDIA CUDA como entorno de trabajo

La idea de calcular las correspondencias entre ambas imágenes captadas por el par estéreo por uno de los métodos nombrados anteriormente, tales como el SAD o el SSD, requiere un coste computacional elevadísimo. Es por ello, que si se piensa utilizar el sistema para el cálculo de distancias a una velocidad razonable (10-30 FPS) se necesitarían numerosos equipos trabajando en paralelo con decenas de CPUs para acercarnos, por lo menos, a los tiempos deseados. No obstante, la opción que pareció más razonable fue la del uso de la tarjeta gráfica como núcleo de procesamiento, por varias razones, algunas ya nombradas anteriormente:

- Mercado en continua evolución creciente en los últimos años.
- Coste muy inferior al de las CPU en términos de €/GFLOPs.
- Tecnología innovadora y con un potencial de desarrollo muy elevado.
- Posibilidad de escalar el sistema sin cambios a nivel de programación en el caso de cambiar la GPU utilizada (o en el caso de utilizar varias simultáneamente).

Una vez que se ha justificado la tecnología a utilizar, en los capítulos restantes se estudiará el problema a resolver, que como ya se comentó, trata de calcular la correspondencia entre un píxel de una imagen con la otra de la forma más eficiente posible y haciendo uso de las posibilidades de aceleración de cálculos que nos ofrece el entorno NVIDIA CUDA.

CAPÍTULO 4. DESARROLLO DEL PROYECTO

A partir de este punto se desarrollarán los algoritmos necesarios para el cálculo de los mapas de disparidades. Para ello se tendrán en cuenta todos los conceptos teóricos explicados en los capítulos anteriores. Además, se irán presentando todas las posibilidades que se adoptaron para la resolución del mismo y el criterio de elección que se tomó para cada una de ellas.

4.1 Código host (CPU)

Para comenzar se hará una breve introducción al modo de cargar imágenes en la GPU que se implementó en dicho código:

El procedimiento será:

1. Carga de las dos imágenes en la CPU.
2. A continuación se reserva memoria para pasar las 2 imágenes al *device* (GPU).
3. Se transfieren los arrays a la memoria de la GPU.
4. Se pasan estos arrays a texturas para optimizar su acceso y también para poder hacer otras operaciones como interpolaciones.
5. Ahora que se tienen las dos imágenes cargadas en la memoria de la GPU, se procede a la creación de varios grids de bloques de hilos, para que se realicen desde la CPU las llamadas a los kernels necesarios.

Las funciones más importantes que se utilizaron para realizar todo lo anterior son:

```
CUT_SAFE_CALL( cutLoadPGMub(image_path, &h_data, &width, &height));
```

Esta función es la que se usa para la carga de la imagen al puntero **h_data** que es de tipo *unsigned char*.

```
CUDA_SAFE_CALL( cudaMallocArray( &cu_array, &channelDesc, width,  
height ));
```

Esta función se usa para reservar memoria en la GPU en un array de dimensiones las de la imagen.

```
CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array, 0, 0, h_data, size,
cudaMemcpyHostToDevice));
```

Esta función se utiliza para pasar la imagen que se encuentra alojada en el array de la memoria de nuestro programa host al array creado anteriormente (**cu_array**).

```
CUDA_SAFE_CALL( cudaBindTextureToArray( tex, cu_array, channelDesc));
```

Esta función se usa para vincular el array alojado en la memoria de la GPU a una textura (**tex**). Previamente se configuró el modo de acceder a la textura a variables tipo *int*, para acelerar su acceso al no tener que realizar operaciones de interpolación, ya que siempre se acceden a coordenadas enteras.

Una vez llegado a este punto se procede a crear el *grid* de bloques de hilos mediante el uso de:

```
dim3 dimBlock(a, b, c);
```

Esta función genera un bloque de hilos de dimensiones (a, b, c) en (x, y, z) respectivamente.

```
dim3 dimGrid(d, e, f);
```

Esta función genera un *grid* de bloques de dimensiones (d, e, f) en (x, y, z) respectivamente.

Una vez se configuran estos parámetros se puede llamar al primer kernel que se ejecutará en el *device*.

4.2 Código device (GPU): kernel para la aplicación de la Laplaciana de la Gausiana

A continuación se detalla el filtro que se programó en el código device, para aplicar la Laplaciana de la Gausiana. Primero se hace una breve reseña teórica para comprender

mejor su funcionamiento, además se irán justificando las elecciones que se tomaron en cada caso.

4.2.1 Eliminación de ruido en una imagen

Como se explica en [4], todas las imágenes tienen una cierta cantidad de ruido, valores distorsionados, bien debidos al sensor CCD de la cámara o al medio de transmisión de la señal. El ruido se manifiesta generalmente en píxeles aislados que toman valores de gris diferentes al de sus vecinos. El ruido lo podemos clasificar en cuatro tipos:

Gausiano: Siempre existe en las imágenes y produce pequeñas variaciones en la misma. Se debe, por ejemplo, a diferentes ganancias en el sensor, perturbaciones en la transmisión, etc. (Figura 4.1).

Por ejemplo, si se toman dos imágenes sucesivas y se restan, se obtendrá el error, que puede describirse como una variable gaussiana.

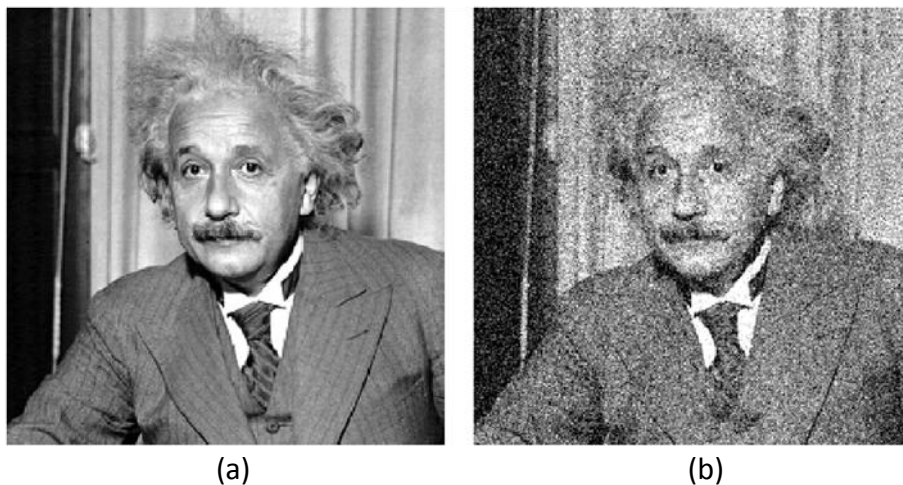


Figura 4.1 Imagen original (a). Ruido gaussiano (b).

Impulsional: Se suele conocer como “sal y pimienta”. Consiste en que, el valor de un píxel no tiene relación con el real, sino con el ruido, que toma valores muy altos o bajos. Entonces se caracteriza porque el píxel toma un valor máximo, causado por una saturación del sensor, o mínimo, si se ha perdido su señal. También se puede dar en

los objetos que se encuentran a altas temperaturas, ya que las cámaras tienen una ganancia alta en el infrarrojo que no dispone el ojo humano (Figura 4.2).

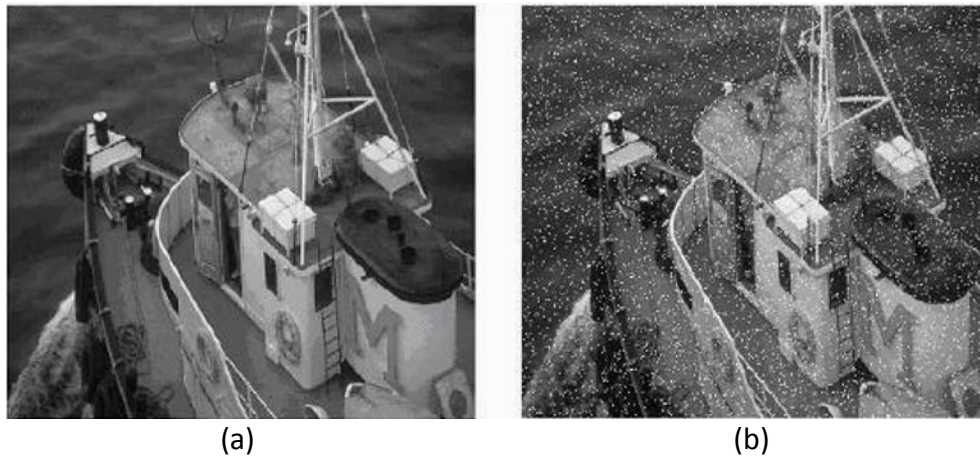


Figura 4.2 Imagen original (a). Ruido impulsional (b).

Frecuencial: La imagen obtenida es la suma entre la ideal y otra señal, la interferencia, que se caracteriza por ser una senoide de frecuencia determinada (Figura 4.3).

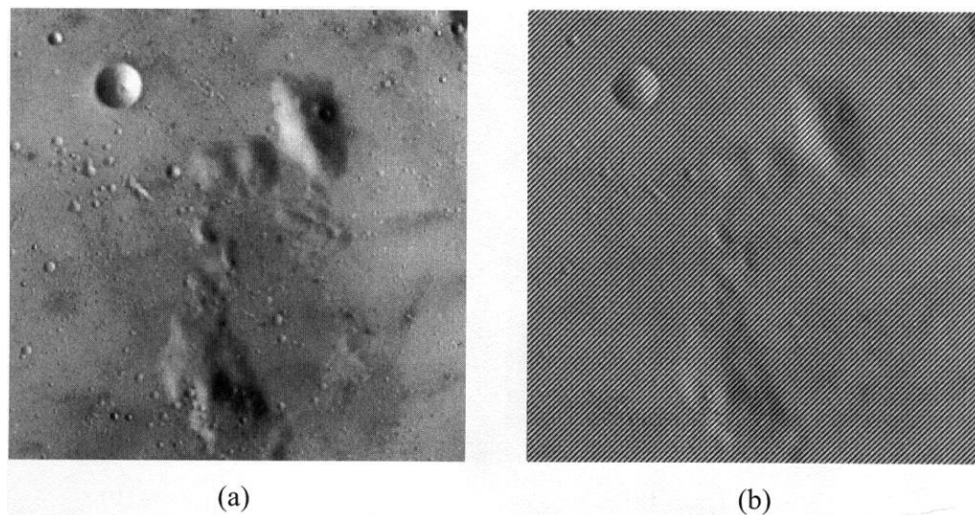


Figura 4.3 Imagen original (a). Ruido frecuencial (b).

Multiplicativo: La imagen obtenida es el resultado de la multiplicación de dos señales (Figura 4.4).

Una de las opciones para la eliminación del ruido es la aplicación de **filtros lineales espaciales**.

Al ser el ruido variaciones de niveles de gris, le corresponden altas frecuencias, por lo que se aplicarán filtros paso bajo para su eliminación.



(a) (b)
Figura 4.4 Imagen original (a). Ruido multiplicativo (b).

Un tipo de filtro paso bajo espacial tiene la forma:

$$k \cdot \begin{bmatrix} 1 & b & 1 \\ b & b^2 & b \\ 1 & b & 1 \end{bmatrix}$$

En los cuales para el cálculo de k se establece que la ganancia del filtro debe ser la unidad, para no variar la de la imagen original, y por lo tanto en este caso será:

$$k = (b^2 + 4b + 4)^{-1} \quad (4.1)$$

Un inconveniente de este tipo de filtros es que, además de eliminar parte del ruido, desdibujan también los contornos y pueden eliminar objetos pequeños.

Otro tipo de máscaras son aquellas que intentan imitar la forma de una gaussiana:

$$G(x, y) = e^{-\frac{(x+y)^2}{2\sigma^2}}$$

Si por ejemplo $\sigma = 0.391$ píxeles la máscara sería:

$$\frac{1}{32} \cdot \begin{bmatrix} 1 & 4 & 1 \\ 4 & 12 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

Existen más tipos de filtros como los **no lineales**, que en particular, se suelen utilizar para eliminar el ruido impulsional, pero no son de importancia para el desarrollo de este proyecto y por lo tanto no se estudiarán.

4.2.2 Detección de bordes en una imagen

Una de las informaciones más útiles que se encuentran en una imagen la constituyen los bordes, ya que al delimitar los objetos, definen los límites entre ellos y el fondo. Las técnicas que se utilizan en la detección de bordes tienen por objeto la localización de los puntos en los que se produce una variación de intensidad. Para ellos se tienen principalmente la posibilidad de utilizar técnicas basadas en la primera (gradiente) o segunda (laplaciana) derivadas.

Las técnicas basadas en el gradiente se basan en el cálculo de diferencias entre píxeles vecinos

Las técnicas de segundo orden se basan en la aplicación de la laplaciana (segunda derivada) y representa la derivada respecto a todas las direcciones.

Las dos máscaras que se utilizan de forma más frecuente para ello son:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Donde el píxel central toma el valor negativo de la suma de todos los que le rodean. Este operador es muy sensible al ruido, por lo que apenas se utiliza.

Otro tipo de métodos son los basados en las derivadas de gaussianas, y en particular, el operador de *Marr-Hildreth*. Éste se basa en la aplicación de una gaussiana (una de las

máscaras que se vieron anteriormente) para eliminar la influencia del ruido y posteriormente se hace la segunda derivada para detectar los bordes.

La ventaja principal de este método es que el contorno detectado es de un solo píxel. Por contra se pueden detectar muchos bordes pudiendo dar lugar al conocido efecto *plato de espagueti*.

4.2.3 Justificación del uso de la Laplaciana de la Gausiana y del método SSD

Para determinar la correlación entre la imagen tomada por la cámara derecha y la cámara izquierda se utilizarán métodos paramétricos. En este caso se decidirá a continuación entre el método SAD (2.17) o el SSD (2.18). El problema de la aplicación de estos métodos (como ya se explico en el CAPÍTULO 2) es que los valores de correlación se ven muy influenciados por la diferencia de ganancia de las cámaras, y por ello, la solución que se adopta es preprocesar la imagen mediante una Laplaciana de Gausiana, de modo que se disminuye la influencia de la iluminación, ganancia de las cámaras y resalta las propiedades de la imagen.

Si se prioriza el coste computacional, intuitivamente puede parecer que el método SAD sea el más adecuado, ya que solo utiliza operaciones de suma/resta. La otra posibilidad era el método SSD (el resto de métodos citados realizan operaciones como raíces cuadradas y por lo tanto se descartaron), ya que únicamente se diferencia del anterior en una operación (realiza una multiplicación en lugar de un valor absoluto).

Una vez se probó el método SSD, los resultados de coste computacional fueron los mismos y en algunos casos hasta algo menores. Esto se debe a que por las características propias de las GPU, el coste computacional de hacer un valor absoluto es prácticamente el mismo a hacer una multiplicación (elevar al cuadrado). Por lo tanto, se escogerá SSD como método para el cálculo de las correspondencias, ya que por norma general ofrece mejores resultados el SAD, aunque se dejará la posibilidad de cambiar a SSA en los parámetros de configuración del programa, como se explicará en el APÉNDICE A.

4.2.4 Justificación del algoritmo implementado

Como se ha explicado anteriormente con la aplicación de dos máscaras (en dos pasadas) de dimensiones 3x3, se podrá llevar a cabo el operador de *Marr-Hildreth* (Laplaciana de una Gausiana), pero se estudió la posibilidad de *fusionarlas* en una de dimensiones 5x5, aunque a priori se realicen más operaciones (25 operaciones por píxel frente a 18) y los resultados indicaron que se tarda menos tiempo haciendo todo de una pasada con esta última máscara. Esto es debido a que es más eficiente esta implementación por el mayor grado de paralelismo alcanzado.

Para la unión de los dos filtros de dimensiones 3x3 en uno de 5x5 se programó un script en MATLAB®, el cual se encuentra en el APÉNDICE B.

Como primera opción, se escogió como gaussiana la de $\sigma = 0.391$ píxeles y la laplaciana en las direcciones de conectividad cuatro (ceros en las esquinas), y como resultado de aplicar dicho script se obtuvo el siguiente filtro:

$$\frac{1}{32} \cdot \begin{bmatrix} 0 & -1 & -4 & -1 & 0 \\ -1 & -4 & 2 & -4 & -1 \\ -4 & 2 & 32 & 2 & -4 \\ -1 & -4 & 2 & -4 & -1 \\ 0 & -1 & -4 & -1 & 0 \end{bmatrix}$$

La otra posibilidad que se planteó fue la elección de la laplaciana con derivada en todas las direcciones (conectividad ocho) y la misma gaussiana que en el caso anterior, y como resultado de aplicar el script se obtuvo el siguiente filtro:

$$\frac{1}{32} \cdot \begin{bmatrix} -1 & -5 & -6 & -5 & -1 \\ -5 & -12 & 10 & -12 & -5 \\ -6 & 10 & 76 & 10 & -6 \\ -5 & -12 & 10 & -12 & -5 \\ -1 & -5 & -6 & -5 & -1 \end{bmatrix}$$

De entre las dos opciones expuestas anteriormente se escogió la primera, por la razón de tener ceros en las esquinas de modo que se requieren menos operaciones por píxel

(cuatro menos), con lo que el tiempo global se disminuirá como se observará a continuación.

Para enfocar la programación de las anteriores convoluciones se propusieron varias alternativas:

- A. Lanzar 25 hilos (tamaño de la máscara) por cada píxel que se desee tratar. Además se seleccionarán distintas dimensiones de bloques de hilos, así como el tamaño del *grid* (nótese que en principio para una GPU ideal sin restricción de memoria así como de hilos a lanzar, se obtendrán mejores resultados lanzando bloques con un número de hilos múltiplo de 16 como ya se explicó en el CAPÍTULO 3). El inconveniente que puede presentar este tipo de enfoque es que se deberán sincronizar los hilos de cada bloque para ser posible el acceso y escritura a la memoria compartida, con lo que se incrementará el tiempo de ejecución.
- B. Lanzar tantos hilos como píxeles tiene la imagen. Cada hilo realizará las operaciones necesarias para la aplicación de dicha convolución.

En la Tabla 4.1 se detallan los tiempos de cómputo entre las diversas posibilidades que se plantearon, sabiendo que el algoritmo se aplicó a una imagen B&W de 640x480 píxeles (VGA).

	Método A con bloques de (5,5,1) y grid de tamaño imagen.	Método A con bloques de (5,5,5) y grid de tamaño imagen.	Método B con bloques múltiplo de 64 hilos (16,16,1).	Método B con bloques de (128,1,1) y grid ajustado.
Tiempo de cómputo (ms)	26	27	1.9	1.5

Tabla 4.1 Comparativa de tiempos entre ambos métodos, utilizando una GPU NVIDIA® GeForce® 8600M GT y un tamaño de imagen de 640x480 píxeles (VGA).

4.2.5 Interpretación de los resultados

Lo primero que se ha de aclarar es que el método que se utilizará será el B, ya que los tiempos son aproximadamente 17 veces menores. Además, después de diversas pruebas se concluyó que la GPU gestiona mejor los bloques de una dimensión que los de dos o tres.

Según [20] es recomendable lanzar bloques múltiplos de 64 hilos para que sea más eficiente la gestión de memoria y por lo tanto los tiempos sean menores. Otra conclusión que se obtuvo, fue que el tiempo era menor cuanto más se acerca el número de hilos a un múltiplo de 64.

Por lo tanto, y en vista de los resultados, se dedujo que lo más óptimo era escoger un bloque unidimensional de un múltiplo del ancho de la imagen.

Una vez llegado a este punto se probaron distintos tamaños para el bloque de hilos, tales como 64, 128, 256 y 512 hilos. Y los resultados que se obtuvieron fueron muy similares en todos los casos. Para el caso de 512 hilos, aunque para la GPU utilizada se pueden lanzar hasta 512 hilos simultáneamente (APÉNDICE E), debido a la memoria necesaria por hilo, no es viable esta solución y se obtuvieron errores en ejecución.

Se hace necesario aclarar, que el tamaño del grid será el correspondiente para tratar la imagen completa y dependerá del tamaño de la misma.

Una vez finalizado este apartado y en vista de las pocas diferencias encontradas entre los distintos tamaños de bloque y de grid, para la aplicación de dicho filtro, se propuso que fuese más funcional el programa y que se auto-ajustase a cualquier tamaño de imagen. Para ello, como posteriormente se verá en el APÉNDICE A, se definió una variable llamada **ANCHO_BLOQUE**, la cual contiene la dimensión **x** del bloque de hilos a lanzar, y el grid lanzará los bloques necesarios para cubrir la dimensión de la imagen a procesar.

De este modo, aunque las diferencias son de menos de 1ms, apenas se verá afectado el resultado final, ya que el tiempo total del cálculo del mapa de disparidad será sobre

dos órdenes de magnitud mayor, y de este modo el software adquirirá una mayor flexibilidad al poder trabajar con distintos tamaños de imagen sin modificar la programación.

4.2.6 Aplicación a dos imágenes

Una vez se ha elegido el tamaño de los bloques y grid, así como el método a desarrollar en el kernel, se hace necesario aplicar todo lo anterior a dos imágenes, que corresponderán a las tomadas por ambas cámaras simultáneamente. Para ello se propusieron varias opciones para implementarlo sobre lo anterior:

1. Dos aplicaciones del mismo kernel, uno para cada imagen.
2. Para cada hilo del kernel anterior se tratarán los píxeles de ambas imágenes.

Los resultados fueron los que se observan en la Tabla 4.2:

	Método 1	Método 2
Tiempo de cómputo (ms)	3.0	2.9

Tabla 4.2 Comparativa de tiempos entre ambos métodos para dos imágenes de 640x480 píxeles, utilizando una GPU NVIDIA® GeForce® 8600M GT.

Como se observa, para el **método 1**, el resultado es el de un kernel individual multiplicado por dos. Mientras que para el **método 2**, el tiempo es menor. Esto es lógico, ya que se debe pensar que el hecho de lanzar hilos tiene un coste computacional y que es apreciable cuando se habla de cientos de miles de hilos, además de tenerse que declarar variables e invocar a otro kernel. Por lo tanto el método que se utilizará será el **método 2**, ya no sólo por el menor tiempo, sino por la sencillez de ampliar el kernel anterior para adaptarlo a otra imagen más.

El resultado de la aplicación de este kernel se puede ver en la Figura 4.5, aunque sólo se muestra una imagen a modo de ejemplo, éste se aplicará para ambas imágenes del par estéreo.



Figura 4.5 Resultado de la aplicación del kernel de la Laplaciana de la Gausiana. Imagen original (a). Laplaciana de la Gausiana (b).

4.3 Código device (GPU): algoritmo para la obtención del mapa de disparidades

Este kernel será el encargado de realizar el cálculo de la correspondencia de cada uno de los píxeles de una imagen con los de la otra. Como se comentó anteriormente, se requiere una enorme potencia de cálculo para llevar a cabo esta búsqueda, y es por ello, por lo que se debe de optimizar al máximo el paralelismo de tareas para optimizar el tiempo de cómputo.

Para simplificar la búsqueda, las cámaras se suponen ya calibradas y ambas imágenes rectificadas, de modo que la búsqueda se reduce a una línea horizontal (línea epipolar).

En la Figura 4.6 se muestra lo que a grandes rasgos realiza este kernel. Tomando por ejemplo la imagen izquierda como referencia, en la derecha se irá desplazando la ventana de búsqueda (en este ejemplo de dimensiones 6x6 píxeles) hacia la izquierda, hasta encontrar la disparidad que ofrezca el mejor valor de correspondencia (11 píxeles de disparidad en el ejemplo).

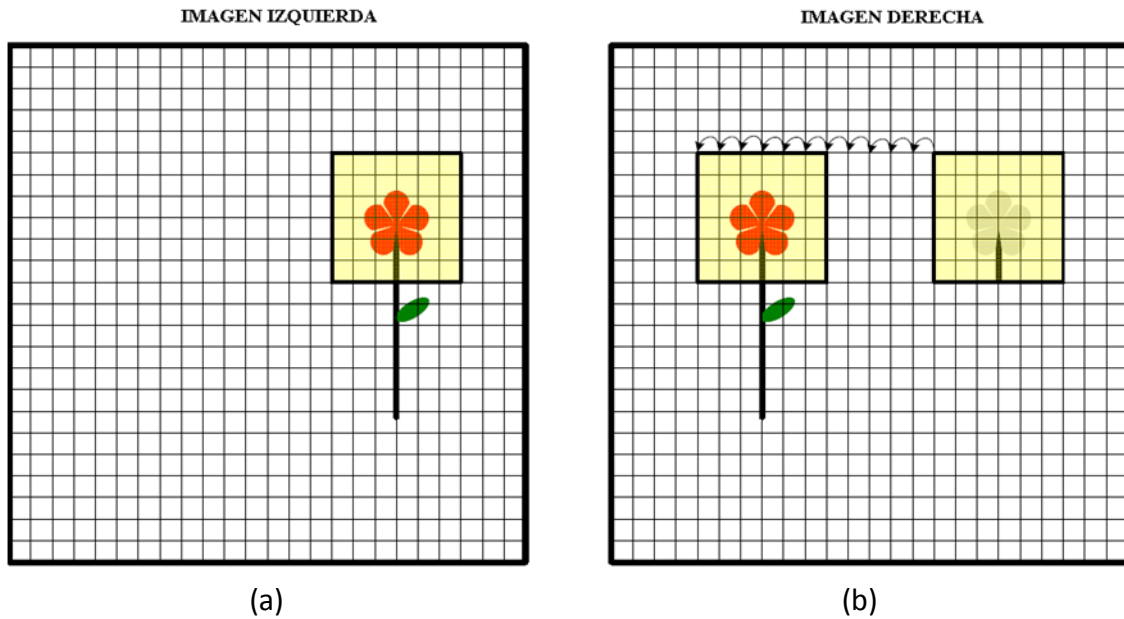


Figura 4.6 Correlación izquierda-derecha. Imagen de referencia (a). Imagen donde se busca (b).

Como se puede observar rápidamente, los requerimientos computacionales para el cálculo de las disparidades estéreo pueden ser enormes. Para reducir los cálculos, se establece un rango de disparidades que suele tener valores típicos entre 50-100 píxeles. No obstante, para una imagen VGA (640x480 píxeles), con una ventana de búsqueda de dimensiones 7x7 píxeles y con un rango de 50 píxeles de disparidad, ¡hay casi 753 millones valores de SSD que se deben de calcular!

Una vez comprendido el coste computacional necesario, queda justificado que a este kernel se le preste una especial atención, ya que será el que ocupe aproximadamente el 99% de los tiempos totales del programa completo. Por lo tanto, a continuación se expondrá de forma más detallada el algoritmo utilizado para el cálculo de la correspondencia.

4.3.1 Procedimiento detallado

El procedimiento que se detalla a continuación es una modificación del descrito en [1]. Para comenzar, se debe elegir donde almacenar las imágenes. Para ello, se usarán texturas a partir de arrays ya creados en el código *host* por varios motivos:

- Mayor velocidad de acceso, ya que son accesos a memoria caché de la GPU.
- No hay problemas aunque se salga fuera de la textura (se configuró el direccionamiento de la textura como *boundary clamping*, mediante el cual aunque se salga de los límites de la misma, se devolverá el valor del borde más próximo).
- Un cambio en la forma de adquisición de las imágenes sería sencillo y no requeriría cambios en el código *device*.

El siguiente paso es determinar cómo se distribuirán los hilos. El enfoque elegido utiliza un hilo para procesar una columna de píxeles, similar al método de Faugeras [21]. Se debe aclarar que, teóricamente, ésta no es la forma más óptima de programar, pero precisamente se utiliza para eliminar operaciones redundantes y minimizar el impacto del tamaño de la ventana de búsqueda en el rendimiento global.

Es importante destacar que las imágenes de entrada de este algoritmo se encuentran procesadas con el filtro de la Laplaciana de la Gaussiana.

Cada hilo acumula la suma de diferencias al cuadrado (o la suma de diferencias absolutas dependiendo de la configuración inicial del algoritmo) de cada columna de píxeles de altura la de la ventana que se esté utilizando. Esta suma de diferencias se calcula entre la imagen de referencia (izquierda) y la imagen de comparación (derecha), para el caso del cálculo de la disparidad izquierda. Se tomará como imagen de referencia la derecha y la de comparación la izquierda, para el caso de la disparidad derecha.

La imagen de comparación se va recorriendo según un offset dado por el valor actual de la disparidad para la que se esté calculando. Es importante que se destaque que los valores de los píxeles se obtienen mediante el uso de texturas, lo cual permite que, aunque en algún momento se pueda salir fuera de los límites de la textura, el valor devuelto será el del borde de la imagen más cercano (ya que se usó el modo apropiado de direccionamiento de la textura para ello ***addressMode = cudaAddressModeClamp***).

La suma de diferencias al cuadrado de cada columna se almacena en memoria local compartida (**extern __shared__**). Debido a que se usan dos píxeles (para el caso de una ventana de 5x5 píxeles), hacia cada lado del bloque que se esté procesando para cumplir con el tamaño de la ventana, el bloque lanzado necesitará cuatro hilos “extra” que dispongan de esas sumas (los hilos 0-3 en la figura de ejemplo realizan una lectura y suma extra).

Una vez que se hayan terminado de realizar las sumas de diferencias al cuadrado para todo el bloque de hilos, entonces cada hilo sumará la columna de valores SSD de sus columnas vecinas (se suma un radio de columnas para la derecha e izquierda) para determinar el valor SSD total para la ventana utilizada. Este valor es el coste, que se comprueba en cada iteración para evaluar el grado de similitud para cada disparidad en el píxel actual.

En la Figura 4.7 y a modo de ejemplo se pueden ver tres píxeles (verde, azul y naranja) con sus respectivas ventanas de búsqueda, marcados con línea discontinua del mismo color.

Si se reescribe la expresión ya vista para el cálculo de la SSD para cada disparidad que se va a calcular se obtiene:

- Para el cálculo de la disparidad izquierda-derecha.

$$SSD_{x,y} = \sum_{i=x-R_H}^{x+R_H} \sum_{j=y-R_V}^{y+R_V} (Izquierda_{i,j} - Derecha_{i-d,j})^2$$

- Para el cálculo de la disparidad derecha-izquierda.

$$SSD_{x,y} = \sum_{i=x-R_H}^{x+R_H} \sum_{j=y-R_V}^{y+R_V} (Derecha_{i,j} - Izquierda_{i+d,j})^2$$

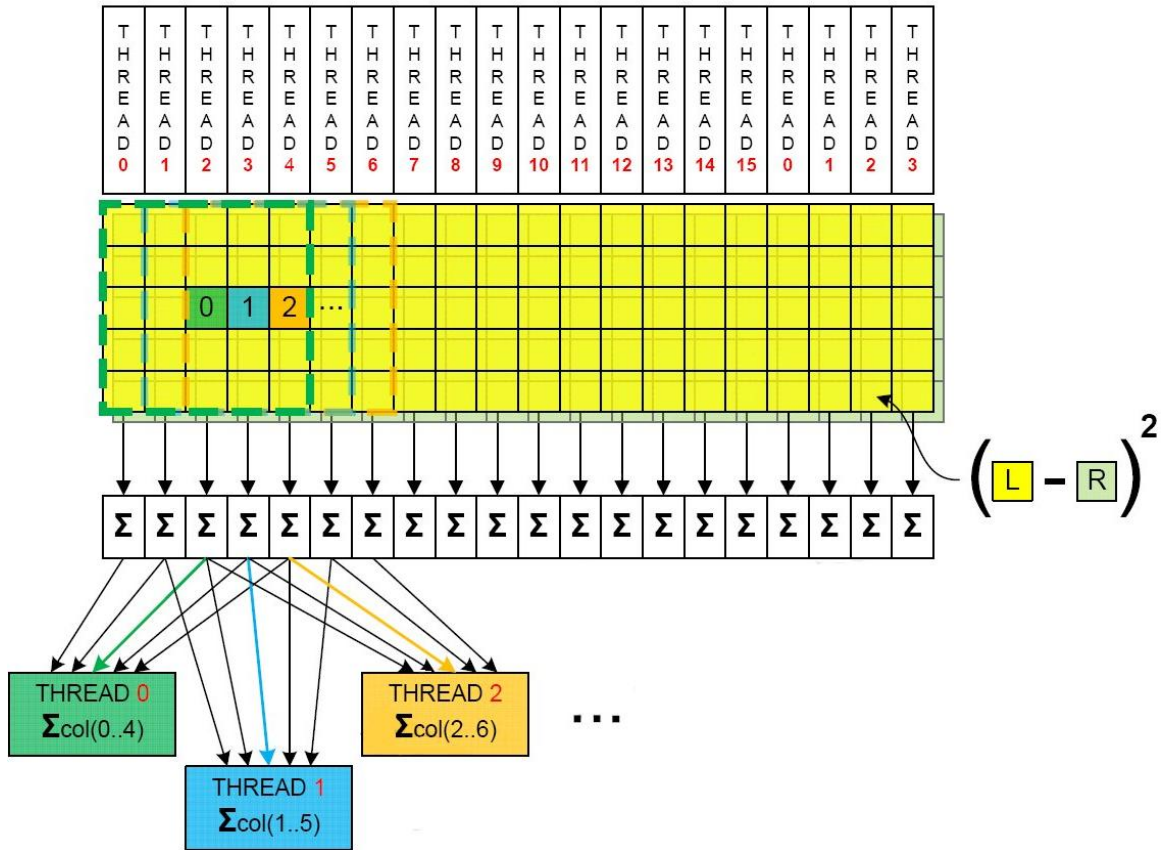


Figura 4.7 Esquema del proceso implementado utilizando una ventana de búsqueda de 5x5 píxeles y un tamaño del bloque de hilos de (16, 1, 1) píxeles.

Donde x, y son las coordenadas del píxel actual, R_H es el radio horizontal de la ventana, R_V es el radio vertical, y por último d es el valor de la disparidad actual evaluado. El radio de la ventana es el número de píxeles a la izquierda, derecha, arriba o abajo del píxel que se está utilizando para la ventana de búsqueda ($R_H = 2$ & $R_V = 2$ para una ventana de 5x5).

Después de que la primera fila de píxeles haya sido procesada, las siguientes filas pueden ser procesadas con menor coste computacional. Una vez que se tiene la suma de todos los valores de una columna para un píxel, si se resta el valor SSD del primer píxel y se suma el de una fila nueva por debajo, se obtiene el valor de la suma para el nuevo píxel.

Como se observa, sólo con dos sumas de diferencias al cuadrado y cuatro lecturas de textura se obtiene la nueva suma de la columna, lo cual reduce notablemente el coste computacional del algoritmo. Además el número de filas que son procesadas por cada hilo se puede modificar en la configuración inicial del programa, como se verá en el APÉNDICE A.

El proceso arriba descrito se repite para cada disparidad (d). En cada paso, el valor actual de coste se compara con el valor mínimo de las sumas SSD hasta el momento. Es importante destacar, que el valor mínimo SSD se almacena en la memoria global en un array del tamaño de la imagen. Si el nuevo valor SSD es menor que el anterior mínimo almacenado, este nuevo valor se convierte en el mínimo y se almacena también con el correspondiente valor de disparidad d en memoria.

Al final del proceso el valor que se obtiene como mejor disparidad se almacena en memoria global y se crea una nueva imagen (**mapa de disparidad**), cuyos píxeles son los valores obtenidos como mejor correspondencia para cada píxel.

4.3.2 Justificación de la solución adoptada

Como se puede observar, este algoritmo utiliza en gran medida la memoria compartida, por ello se optimiza la programación en paralelo de forma que todos los hilos están cooperando entre sí.

Además, las columnas de valores SSD se almacenan en memoria compartida, lo que posibilita su disposición instantánea a todos los hilos del bloque. Esta forma de compartir numerosos datos entre hilos hace mejorar mucho el rendimiento. En vez de que los valores SSD del kernel sean calculados por un hilo, lo que se hace es que se coopere entre todos los hilos para obtener la suma del valor total SSD del kernel. Los hilos usarán las columnas de sumas vecinas para completar el valor SSD del kernel.

Si se usara un kernel de 5x5 píxeles sin cooperación entre hilos, se requerirían 25 diferencias cuadráticas y 24 sumas por cada píxel. En este ejemplo, después de calcular la primera fila sólo se necesitan 2 diferencias cuadráticas y 4 sumas para calcular el valor del nuevo píxel. Para un kernel mayor de 11x11 píxeles, se necesitan 2

diferencias cuadráticas y 10 sumas, mientras que de la forma tradicional se requerirían 121 diferencias cuadráticas y 120 sumas, lo que supone una reducción de 20 veces el número de operaciones!

CAPÍTULO 5. RESULTADOS

En este capítulo se muestran los resultados obtenidos, así como la interpretación de los mismos.

Hay que destacar que todas las imágenes utilizadas se encuentran en escala de grises y en formato PGM, se obtuvieron de [22], todas ellas ya se encuentran rectificadas, además de suponerse que ambas cámaras ya se encuentran calibradas para la realización de todas las pruebas.

Además, la configuración inicial del algoritmo se establece a la óptima para la GPU NVIDIA® GeForce® 8600M GT, con la que se han realizado todas las pruebas, tal y como se explica en el APÉNDICE A.

5.1 Gráficas de tiempos de cómputo

En la Figura 5.1 se pueden apreciar los resultados que se obtuvieron experimentalmente. Se realizaron ensayos para un numeroso rango de disparidades, así como de dimensiones de ventana, entre los que se encuentran los que habitualmente se utilizarán para el caso de un sistema real.

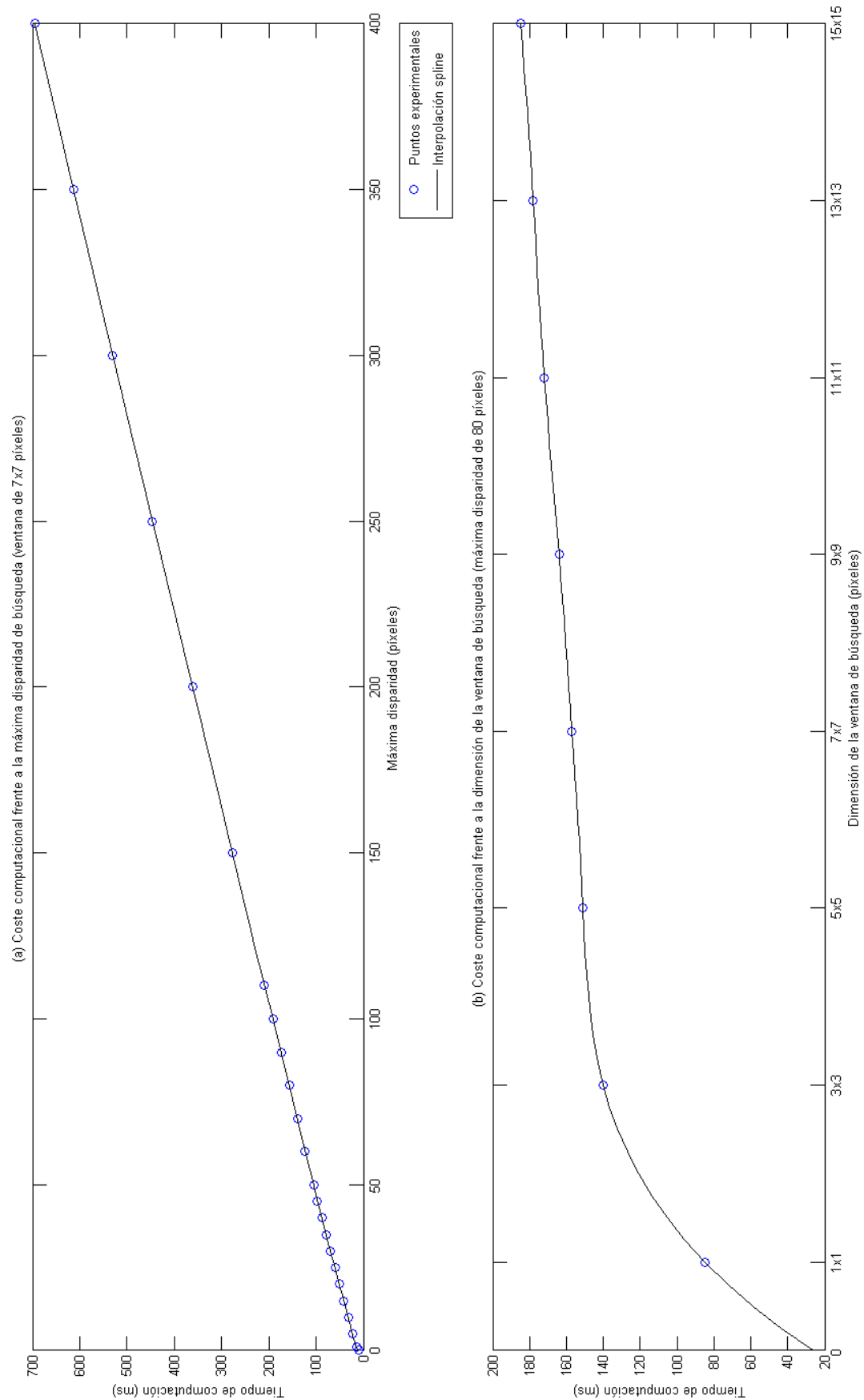


Figura 5.1 Gráficas de tiempos de cómputo en función de la máxima disparidad de búsqueda (a) y el tamaño de la ventana de búsqueda (b). Ambas realizadas con NVIDIA® GeForce® 8600M GT y la configuración óptima del algoritmo para dos imágenes de 640x480 píxeles.

5.1.1 Interpretación de las gráficas

En este apartado se interpretan ambas gráficas de la Figura 5.1. Para ello se explica la configuración inicial utilizada, además de las pertinentes conclusiones de las mismas.

5.1.1.1 Coste computacional frente a la máxima disparidad de búsqueda

La configuración inicial del algoritmo con la que se realizaron las pruebas fue la siguiente:

```
//*****

#define ROWSperTHREAD 40

#define BLOCK_W 64

#define ANCHO_BLOQUE 64

#define RADIUS_H 3

#define RADIUS_V 3

#define MIN_SSD 500000

#define STEREO_MIND 0.0f

#define STEREO_MAXD LA VARIABLE QUE SE MODIFICA

#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )

#define VALOR_INICIAL 255

//*****
```

Según se observa en la Figura 5.1 (a) el tiempo de cómputo crece con la máxima disparidad y lo hace de forma lineal, lo cual es lógico, ya que el algoritmo programado no varía su grado de paralelismo al modificar dicha variable. Esto es así porque las tareas de búsqueda se repiten para cada hilo tantas veces como disparidad máxima esté configurada en el algoritmo.

Como conclusión de esta gráfica, se tiene que la elección de la máxima disparidad a utilizar será crítica en el tiempo global de cómputo y por lo tanto, se deberá ajustar en base a los siguientes objetivos (previamente se fijará el tamaño de la ventana deseado, por defecto es de 7x7):

- Se determinará según los requerimientos la distancia mínima a la que interesa detectar un objeto. Para ello se utilizarán los parámetros de la cámara, como el tamaño del píxel, distancia focal, etc. y con ello se podrá establecer una relación entre píxeles de disparidad y distancia al objeto. Por lo tanto, se utilizará esta disparidad como máxima en la configuración inicial del algoritmo.
- Una vez se ha establecido el valor de máxima disparidad, si se tiene el caso de que el tiempo de cómputo sea inaceptable, se procederá del siguiente modo:
 - Se realizará el cálculo de la distancia máxima hasta la cual interesa detectar objetos con precisión. A continuación se calculará la resolución mínima que debe tener la imagen para cumplir con dicha distancia. Una vez calculada esta resolución, si es menor que la utilizada, se podrá proceder a disminuir la resolución de captura, hasta conseguir tiempos aceptables de cálculo.

5.1.1.2 Coste computacional frente a la dimensión de la ventana de búsqueda

La configuración inicial del algoritmo con la que se realizaron las pruebas fue la siguiente:

```
//*****  
  
#define ROWSperTHREAD 40  
  
#define BLOCK_W 64  
  
#define ANCHO_BLOQUE 64  
  
#define RADIUS_H LA VARIABLE QUE SE MODIFICA  
  
#define RADIUS_V LA VARIABLE QUE SE MODIFICA
```

```
#define MIN_SSD 500000

#define STEREO_MIND 0.0f

#define STEREO_MAXD 80.0f

#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )

#define VALOR_INICIAL 255

//*****
```

Según se observa en la Figura 5.1 (b), a partir de las dimensiones 3x3 el tiempo crece de manera prácticamente lineal. Dimensiones iguales o menores de 3x3 apenas se consideran para la aplicación en casos prácticos, ya que los resultados de la búsqueda pueden tener numerosos errores.

Es muy importante destacar que con una programación convencional el tiempo crecería enormemente al aumentar la dimensión de la ventana (el número de elementos crece cuadráticamente con dimensión), pero debido a la programación tan eficiente que se implementó y aprovechando las ventajas del procesamiento paralelo esto no es así.

Este comportamiento de la curva es el que demuestra la tremenda eficiencia del algoritmo implementado, que no sólo ofrece tiempos pequeños para una ventana determinada, sino que no varían demasiado al aumentar el tamaño de la misma. Para llegar a este grado de paralelismo es necesario ajustar los parámetros como se explicará en el APÉNDICE A (ya realizado en las gráficas anteriores).

Por lo tanto, si observamos el comportamiento de la curva, se puede concluir que una vez que se ha elegido el valor de la máxima disparidad, podremos ir probando con las distintas dimensiones de ventana para cual obtenemos mejores resultados. En general, para zonas con bajas texturas, se necesitarán tamaños grandes de ventana, mientras que para zonas con texturas bastará con tamaños pequeños de ventana. En vista de la gráfica, puede parecer que al no suponer mucha diferencia en cuanto a coste computacional, elegir dimensiones más grandes sea lo más adecuado. Pero esto

tiene sus problemas, ya que a medida que aumentamos el tamaño, se van perdiendo detalles en el mapa de disparidad, debido a errores en zonas de discontinuidades de profundidad. Por lo tanto, ajustar este valor es una tarea delicada y dependerá de numerosos factores, como el tamaño de los objetos a detectar, tipo de texturas de los objetos, precisión deseada en cuanto a detalles, etc. Por lo tanto, se recomienda la realización de pruebas para llegar a situaciones de compromiso entre los detalles detectados y los errores cometidos.

5.2 Mapas de disparidades obtenidos

A partir del par de imágenes estéreo ya rectificadas que se observan en la Figura 5.2, se detallan diversos resultados en función de la dimensión del tamaño de la ventana de búsqueda, así como de la limitación de disparidad máxima utilizada.

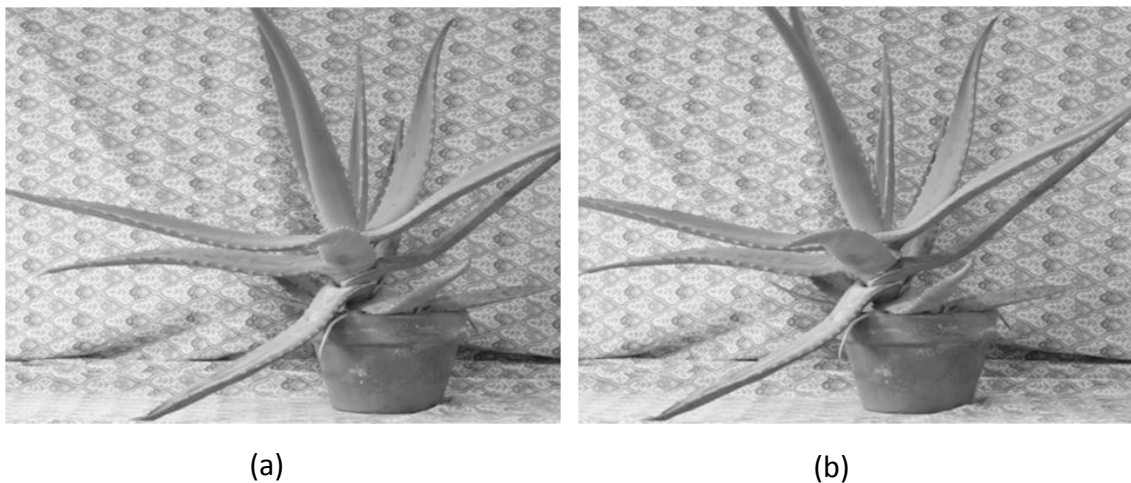
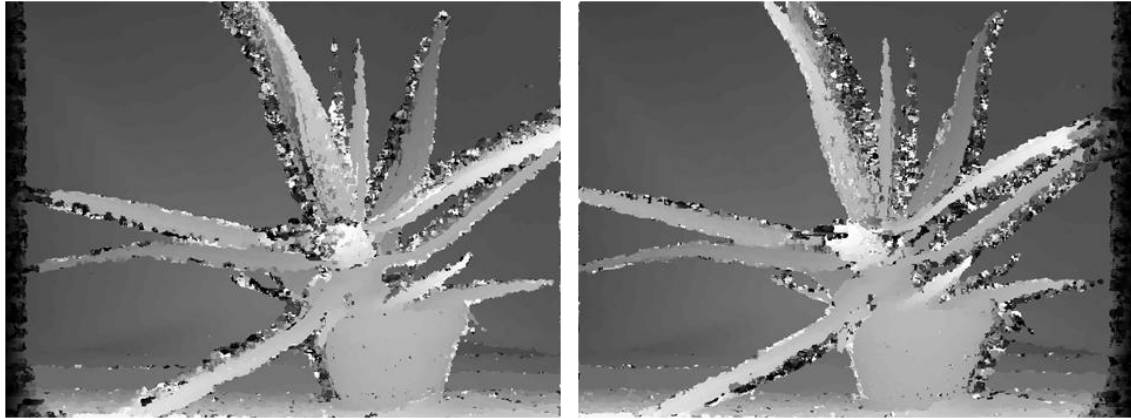


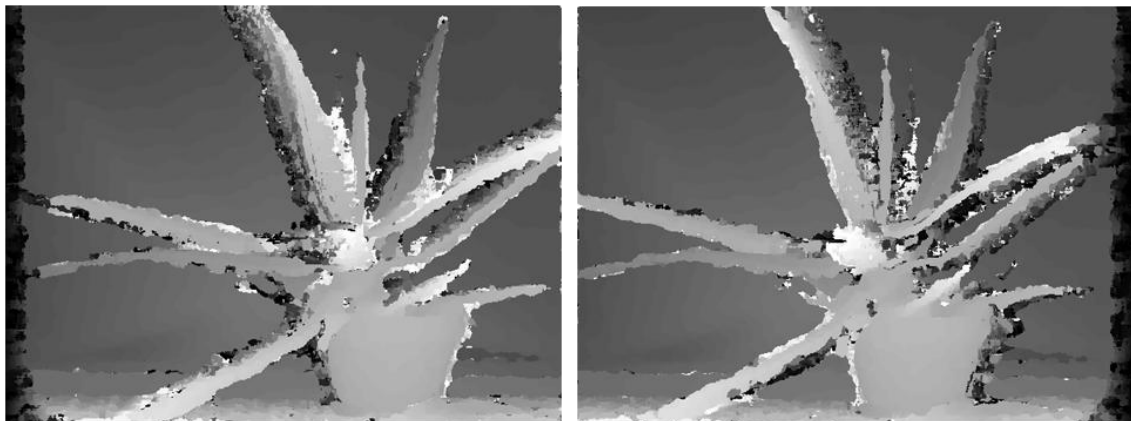
Figura 5.2 Imágenes originales de entrada al algoritmo ambas de resolución 640x480 píxeles. Imagen izquierda ya rectificada (a). Imagen derecha ya rectificada (b).



(a)

(b)

Figura 5.3 Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 7x7 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 155msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA ® GeForce® 8600M GT.



(a)

(b)

Figura 5.4 Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 11x11 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 169msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA ® GeForce® 8600M GT.

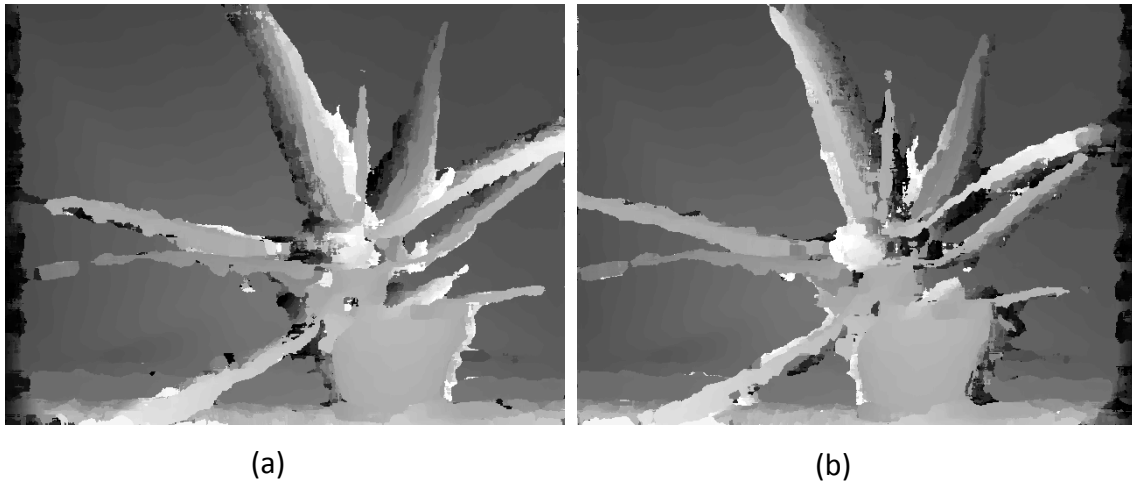


Figura 5.5 Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 17x17 píxeles y una limitación de máxima disparidad de 80 píxeles. El tiempo de cómputo fue de 189msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA® GeForce® 8600M GT.

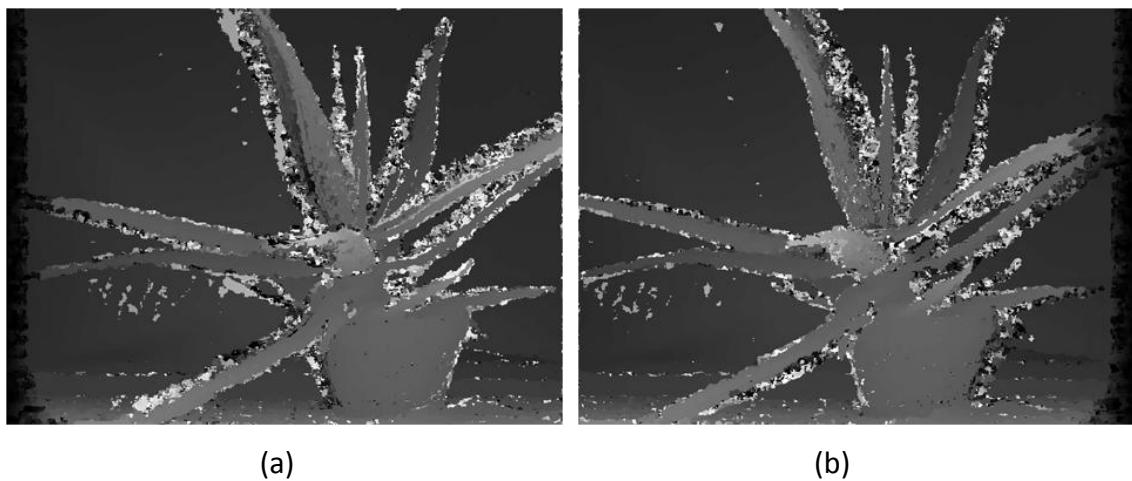


Figura 5.6 Mapa de disparidades izquierda (a) y derecha (b). Se utilizó una ventana de búsqueda de 7x7 píxeles y una limitación de máxima disparidad de 150 píxeles. El tiempo de cómputo fue de 272msec para ambas imágenes de 640x480 píxeles con la GPU NVIDIA® GeForce® 8600M GT.

5.2.1 Comentarios acerca de los resultados

Lo primero que se debe explicar es que todos los resultados se obtuvieron con el método de correspondencia SSD.

Además, se puede apreciar que en todas las imágenes hay zonas con mucha cantidad de textura, por lo que el algoritmo funciona muy bien para prácticamente cualquier tamaño de ventana (con 7x7 podría ser suficiente para muchas aplicaciones). Respecto a la disparidad máxima a utilizar, con 80 funciona bien. Tal vez algunas de las zonas más cercanas a la cámara tengan un poco de error (posible caso de tener en realidad disparidades mayores de 80), pero no merece la pena incrementar el coste computacional para arreglar unas zonas que podrían ser menores del 1% de la imagen en este caso (Figura 5.6).

Para casos de numerosas y diferentes texturas, a distancias similares a las de esta imagen, una configuración similar a la usada en la Figura 5.3 (ventana de 7x7 y máxima disparidad de 80 píxeles) sería adecuada para empezar a ajustar el algoritmo. En otras aplicaciones que admiten menor error, porque, por ejemplo, no se tiene un posterior filtrado del mapa de disparidades, se podría utilizar la configuración de la Figura 5.5 (ventana de 17x17 y máxima disparidad de 80 píxeles), que con muy poco incremento del tiempo de cómputo permite obtener un mapa de disparidades con muchos menos errores (siempre y cuando la imagen tenga zonas de mucha textura).

Para concluir con este capítulo, se debe añadir que la configuración más óptima del algoritmo dependerá en gran medida de las imágenes a analizar. Por lo tanto, se recomienda una exhaustiva calibración del mismo para llegar a una situación de compromiso entre precisión del mapa de disparidades y tiempo de cómputo total.

CAPÍTULO 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 Cumplimiento de objetivos

El objetivo principal de este proyecto era la obtención de mapas de disparidad a partir de imágenes estéreo, minimizando el tiempo de cómputo a la vez que se hacía un programa flexible en cuanto a parámetros modificables y a ampliaciones futuras. Ambos requisitos se han cumplido, buscando siempre soluciones de compromiso para satisfacer los tiempos de cálculo. Para ello, se presentaron diversas soluciones entre las que se eligió siempre la mejor que se amoldaba a cada necesidad.

Por último, es importante destacar que la programación de los algoritmos que se ejecutan en la GPU (*kernels*) se hizo de forma modular, por lo cual se adapta fácilmente a futuras ampliaciones, así como a la integración en un software más amplio de visión estéreo.

6.2 Implementación en un sistema real

Todo lo que se ha desarrollado en este proyecto tiene como objetivo final su implementación en el nuevo vehículo experimental de la UC3M llamado IVVI 2.0 (Figura 6.1 y Figura 6.2) (sucesor del antiguo IVVI).



Figura 6.1 Vista lateral del IVVI 2.0.



Figura 6.2 Vista frontal del IVVI 2.0.

Este vehículo es idóneo para la aplicación de los algoritmos desarrollados durante este proyecto, debido principalmente a las limitadas capacidades de cálculo del mismo, por lo que los sistemas que se implementan en dicho vehículo requieren de una exhaustiva optimización de los tiempos de cómputo, además de requerir de aplicaciones que se acerquen al funcionamiento en tiempo real.

Por otra parte, en este vehículo se encuentra instalado un sistema estéreo de cámaras, el cual se puede apreciar en la Figura 6.3.



Figura 6.3 Sistema estéreo del IVVI 2.0.

Este vehículo dispone de tres ordenadores situados en el maletero (Figura 6.4) que analizan la información sensorial, además pueden intercambiar información entre sí ya que se encuentran conectados en red. El usuario los podrá controlar mediante la pantalla situada en el salpicadero (Figura 6.5), de forma que se hace más sencilla la tarea de modificar los programas ya introducidos.



Figura 6.4 Ordenadores Apple situados en el maletero del IVVI 2.0.



Figura 6.5 Monitor de control en el IVVI 2.0.

Actualmente se está trabajando en la implementación de estos algoritmos de visión estéreo en el sistema estéreo de este vehículo, lo cual indica la versatilidad de los algoritmos desarrollados, así como la flexibilidad para adaptarlos a un sistema mayor.

6.3 Líneas futuras de investigación

A continuación se detallan una serie de puntos que se podrían mejorar en versiones futuras de este mismo algoritmo para el cálculo de mapas de disparidad, ya que debido a demasiados requerimientos computacionales, o por no ser de interés para la aplicación actual, no se han implementado.

Los métodos de correspondencia utilizados (SSD ó SSA) fallan cuando hay poca textura, es por ello por lo que se podría pensar en la combinación de algoritmos de búsqueda de esquinas, bordes, etc. con los métodos utilizados para llegar a resultados bastante más fiables. Por lo tanto, si la zona analizada contiene poca textura, mediante unas evaluaciones previas, se podría, por ejemplo, dividir en ventanas más pequeñas de imagen. Una vez llegado a este punto, para cada ventana se podría calcular la cantidad de textura, para establecer un tamaño de ventana adecuado.

En caso de que la precisión con la que se desean obtener las distancias sea mayor, además de lo anterior, se propone la variación del tamaño de la ventana de búsqueda en función de las necesidades para la zona analizada. A grandes rasgos, lo que se propone es incrementar el radio de la ventana, hasta que la desviación estándar de las tonalidades en los píxeles dentro de ella sea mayor a un umbral dado [23] (de modo que se asegure la suficiente textura dentro de la ventana). Para la Figura 6.6 en la Figura 6.7 se aprecia la variación que sufriría el tamaño de la ventana en una imagen.



Figura 6.6 Par de imágenes estéreo rectificadas.

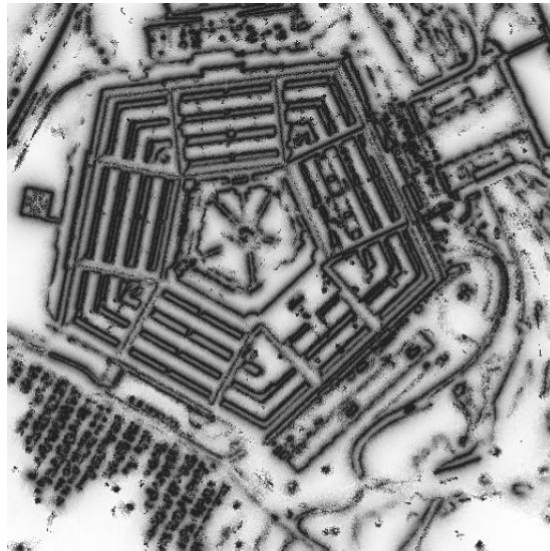


Figura 6.7 En esta imagen el tono de cada píxel indica el tamaño utilizado para la ventana de búsqueda (tonos más claros indican un mayor tamaño de ventana).

Los métodos descritos anteriormente tienen la pega de tener que reprogramar todo el algoritmo, por lo que el coste sería muy elevado y por lo tanto serían soluciones casi inviables si no se dispone del potencial necesario para su reprogramación. Además sería muy complicado paralelizar las tareas de forma efectiva.

De forma más general, si tenemos zonas con poca textura, se pueden intentar corregir los errores, en la medida de lo posible, aumentando el tamaño de ventana, como se explicará en el APÉNDICE A.

Por otro lado, si el hardware disponible tuviera recursos sobrados para el cálculo de las disparidades, para los tiempos deseados se podría plantear el uso de precisión subpíxel, siempre y cuando se requiriese más precisión en el mapa de disparidades.

En el mapa de disparidad que se obtiene como salida de este algoritmo, los valores se encuentran limitados a valores enteros de píxeles, lo que ocasiona que los cambios de disparidad no sean suaves, se obtenga un mapa tosco y no se aprecien los detalles cuya profundidad esté en disparidades intermedias. Como ya se explicó, las zonas próximas a los valores de máxima correspondencia describen una parábola, cuyo valor máximo se puede calcular como se expresó en el CAPÍTULO 2. Por lo tanto, como resultado se obtendría lo que se puede observar en la Figura 6.8.

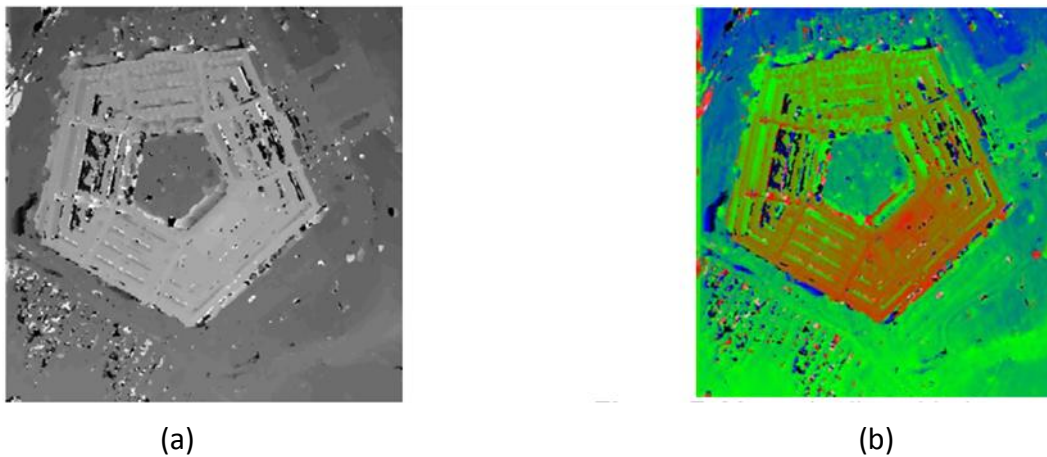


Figura 6.8 Mapa de disparidades con precisión de píxel (a). Mapa de disparidades con precisión subpíxel y con falso color (b).

Otro de los puntos que se podría optimizar, llegado el caso, sería el algoritmo de cálculo de la Laplaciana de la Gaussiana. Se podría aumentar el paralelismo entre los hilos mediante el uso de memoria compartida [24], cosa que a día de hoy es poco conveniente, ya que en términos del tiempo de cómputo total del mapa de disparidad apenas llega al 1%.

En cuanto a la salida del algoritmo, sería conveniente la programación de algún método de filtrado de los mapas de disparidades. Por la forma de programación modular y para no perder el grado de paralelismo actual, además del criterio de mínimo coste computacional, el método de *cross-checking* sería el más adecuado.

Con la evolución de las GPU en el mercado actual se recomienda, con el paso del tiempo, la modificación del algoritmo de cálculo del mapa de disparidades para incluir los procedimientos arriba empleados, y llegado el caso, también se podrían implementar otros métodos de cálculo de correspondencias, que anteriormente se descartaron por los requisitos computacionales requeridos, como es el caso de la correlación normalizada.

CAPÍTULO 7. COSTES DEL PROYECTO

En este capítulo se presentan de forma aproximada los costes del proyecto que han sido necesarios. Para ello se desglosan las tareas que se realizaron ordenándolas temporalmente.

En la Tabla 7.1 se observan los costes de tiempo para cada fase del proyecto, mientras que en la Tabla 7.2 se detallan los costes de material como de tiempo en términos monetarios.

<u>FASES</u>	<u>TIEMPO (h)</u>
Estudio y comprensión del problema a resolver	10
Pruebas preliminares de iniciación a CUDA	10
Implementación del código	60
Pruebas al algoritmo	20
Corrección de errores	10
Pruebas al algoritmo	10
Mejoras para optimizar el código (ajuste de parámetros, ajustes iniciales, etc.)	60
Pruebas al algoritmo	10
Mejoras en la estructura del código	10
	SUBTOTAL1 = 200h
Memoria del proyecto (realización paralela al mismo)	100
Manual de instrucciones anexo a la memoria	5
	SUBTOTAL2 = 105h
	<u>TOTAL = 305h</u>

Tabla 7.1 Tiempo invertido en cada fase del proyecto.

Suponiendo un precio horario aproximado de 30€/hora para un programador novel, el coste total sería:

<u>CONCEPTO</u>	<u>COSTE</u>
Coste asociado a las horas ingeniero.	$305h \cdot 30 \frac{\text{€}}{h} = 9150\text{€}$
Materiales: PC con tarjeta gráfica compatible con CUDA	1000€
Software utilizado: Compilador C++ versión gratuita, librerías y drivers para compilar NVIDIA® CUDA™.	0€
	<u>TOTAL = 10150€</u>

Tabla 7.2 Costes económicos del proyecto.

APÉNDICE A. MANUAL DE USUARIO

En este capítulo se detallan las opciones modificables del programa tanto para el código ejecutable en CPU (*host*) como para el de la GPU (*device*). Además, junto a cada variable, se detalla la forma de ajustarla de forma óptima para su correcto funcionamiento.

A.1 Opciones modificables del código host (CPU)

```
//*****  
  
char *image_filename = "izquierda.pgm";  
  
char *image_filename2 = "derecha.pgm";  
  
char *output_name = "SALIDA_IZQUIERDA.pgm";  
  
char *output_name2 = "SALIDA_DERECHA.pgm";  
  
//*****
```

Lo primero que se deberá configurar son los nombres de las imágenes de entrada/salida. Como se observa, el formato de todas es PGM y los nombres por defecto son los que se muestran. En principio, el algoritmo se encuentra configurado de forma que sólo toma dos imágenes almacenadas en el disco duro (en la carpeta */data* dentro de la ruta del proyecto), pero para su uso en un sistema real, se utilizaría como fuente de entrada las imágenes proporcionadas por el par estéreo.

A.2 Opciones modificables del código device (GPU)

```
//*****  
  
#define SQ(a) (__mul24(a,a))  
  
#define ROWSperTHREAD 40  
  
#define BLOCK_W 64
```

```
#define ANCHO_BLOQUE 64

#define RADIUS_H 3

#define RADIUS_V 3

#define MIN_SSD 500000

#define STEREO_MIND 0.0f

#define STEREO_MAXD 80.0f

#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )

#define VALOR_INICIAL 255

//*****
```

En el código device se encuentran todas estas opciones arriba mostradas, a continuación se explicará la función de cada una de ellas:

ROWSperTHREAD: Esta variable hace referencia al número de filas que calculará cada hilo, es decir, al número de píxeles en vertical para los cuales calculará la disparidad que ofrezca la mejor correspondencia. En principio a medida que aumentamos su valor por encima de 1, para calcular el valor total SSD no necesitará realizar todas las sumas, ya que usará las previas de los píxeles superiores añadiendo un valor y quitando otro para disminuir el coste computacional.

Puede parecer que cuanto mayor sea este valor (límite el alto de la imagen) menor será el tiempo total de cómputo. Esto no es así, ya que para los códigos de elevado paralelismo hay que tener en cuenta la cantidad de veces que accedemos a memoria por cada hilo. Es por ello por lo que requiere ser ajustado.

- Ajuste de su valor: Se establecerá un valor de 40 y sólo después de tener ajustado ANCHO_BLOQUE se procederá a modificar sus valores. Se irá aumentando su valor hasta ver que no disminuya el tiempo de cómputo global. Para el caso de no encontrarse valores que lo mejoren, se probará a disminuir su valor para proceder de forma análoga hasta encontrar un punto óptimo.

BLOCK_W: Esta variable hace referencia al número de hilos en la dimensión x que contiene cada bloque del grid (los bloques son 1D). Para su ajuste, según las especificaciones de las tarjetas gráficas NVIDIA, debe ser un múltiplo de 32 (warp size, que es el número de hilos que se ejecutan simultáneamente), pero por motivos de organización de tareas por parte del compilador, son recomendables múltiplos de 64.

- Ajuste de su valor: Será el primer valor a ajustar y se procederá a probar primero con 32, si no se mejora el tiempo de cómputo se irá aumentando su valor en múltiplos de 32 hasta que el compilador nos de un error de ejecución, ya que al aumentar el número de hilos, la memoria disponible para cada uno de ellos disminuye, y por lo tanto llegará un momento en el que no se disponga de la cantidad necesaria utilizada. El valor final será el que menor tiempo de cómputo nos proporcione.

ANCHO_BLOQUE: Se procederá de forma análoga a la variable anterior, aunque para este caso apenas se apreciará la diferencia. Esto es debido a que ahora se están modificando sólo los hilos que se lanzarán al kernel que aplica la Laplaciana de la Gaussiana y su contribución al tiempo global es mínima (en torno al 1% del tiempo total de cómputo). Por lo tanto su ajuste es apenas necesario y se podría obviar.

RADIUS_H & RADIUS_V: Son las variables que hacen referencia a las dimensiones de la ventana de búsqueda que se utilizará.

- Ajuste de su valor: Como se observa, los valores a introducir son el radio horizontal y vertical de la ventana en píxeles (por ejemplo si se desea utilizar una ventana de 11x11 se establecerán ambos valores a 5). Por defecto, se encuentran ambos valores a 3, lo que es igual a una ventana de 7x7 píxeles.

MIN_SSD: Esta variable hace referencia al valor mínimo que se establecerá como válido para las correspondencias encontradas, es decir, a partir del cual la correspondencia encontrada será descartada.

- Ajuste de su valor: Es necesario que su valor sea menor o igual al máximo que se puede encontrar para la ventana utilizada, que se obtiene de aplicar las

siguientes expresiones y que dependen del método de correspondencia utilizado:

$$\begin{aligned} MIN_SSD &\leq 255^2 \cdot (2 \cdot RADIUS_H + 1) \cdot (2 \cdot RADIUS_V + 1) \\ MIN_SSA &\leq 255 \cdot (2 \cdot RADIUS_H + 1) \cdot (2 \cdot RADIUS_V + 1) \end{aligned} \quad (A.1)$$

Su valor dependerá de lo que el usuario considere una correlación válida para la aplicación deseada. Si no se desea modificar su valor por falta de cálculos, se recomienda establecer su valor al máximo de las expresiones anteriores.

STEREO_MIND: Esta variable hace referencia al valor mínimo de disparidad a partir del que se empezará a buscar las correspondencias.

- Ajuste de su valor: Por defecto está en 0, pero si a conciencia del usuario considera que el infinito de distancias está en un valor de α píxeles de disparidad entre ambas imágenes puede modificarlo. Los objetos más lejanos (menores disparidades) no se podrán identificar, y por lo tanto sólo se recomienda su modificación para usos concretos y a conciencia de los riesgos que pueda conllevar. Si aún así se quiere modificar, se recomienda hacer unos cálculos de equivalencia entre distancia y píxeles de disparidad.

STEREO_MAXD: Esta variable hace referencia al valor máximo de disparidad a partir del cual ya no se buscarán más correspondencias.

- Ajuste de su valor: El rango típico en aplicaciones estéreo suele ser de 50-100, pero su valor concreto dependerá de la aplicación en la que se utilice el programa. Principalmente dependerá del valor mínimo de distancia al que interese detectar objetos, ya que los objetos más cercanos (y que por lo tanto tengan mayores disparidades entre ambas imágenes) no serán detectados con precisión. Se recomienda para modificar su valor, hacer unos cálculos de equivalencia entre distancia y píxeles de disparidad.

SHARED_MEM_SIZE: Esta variable hace referencia a la cantidad de memoria compartida utilizada, y no se deberá cambiar a no ser que se desee modificar el kernel que lleva a cabo la obtención de la disparidad, y añadir alguna variable adicional a compartir entre hilos.

VALOR_INICIAL: Será al que se inicializará el mapa de disparidad de salida del algoritmo, por lo tanto será el que permanecerá en el caso de no haberse obtenido ningún valor de correspondencia inferior al establecido en MIN_SSD.

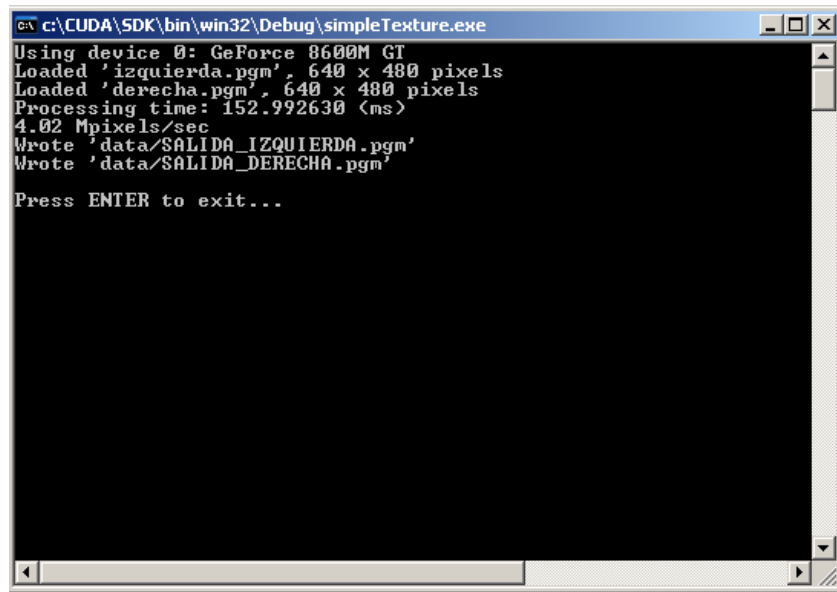
- Ajuste de su valor: Se recomienda utilizar un valor de 255, ya que en el caso de no encontrar correspondencias adecuadas (menores que MIN_SSD) el algoritmo devolverá un valor de disparidad máxima, es decir, que el error será hacia la proximidad de un objeto. Por lo tanto, siempre se tenderá a acercar los objetos en caso de error, lo cual es de tremenda importancia para el caso de detección de obstáculos por ejemplo.

SQ(a) (MÉTODO): Esta definición es la que permite modificar el método de correspondencia que se utilizará.

- Ajuste de su valor: Si en método se escribe **__mul24(a,a)** se implementará el SSD (suma de diferencias cuadradas). Si se escribe **abs(a)** se implementará el SSA (suma de diferencias absolutas).

Se recomienda el uso de SSD, ya que como se justificó anteriormente, además de proporcionar mejores resultados, el coste computacional es igual o algo inferior al del método SSA.

Para terminar con este apartado, a continuación se muestra un detalle de la consola una vez ejecutado el programa, en la cual se observa: modelo de la tarjeta gráfica utilizada, nombre y tamaño de las imágenes cargadas, tiempo de cómputo, Mpixel/s procesados y el nombre de las imágenes de salida.



```
c:\CUDA\SDK\bin\win32\Debug\simpleTexture.exe
Using device 0: GeForce 8600M GT
Loaded 'izquierda.pgm', 640 x 480 pixels
Loaded 'derecha.pgm', 640 x 480 pixels
Processing time: 152.992630 (ms)
4.02 Mpixels/sec
Wrote 'data/SALIDA_IZQUIERDA.pgm'
Wrote 'data/SALIDA_DERECHA.pgm'
Press ENTER to exit...
```

Figura A.1 Consola de resultados de ejecución del programa.

APÉNDICE B.SCRIPT DE MATLAB

En esta sección se adjunta el script de MATLAB® que se utilizó para unir dos convoluciones de dimensiones 3x3 en una sola de dimensiones 5x5. Debe destacarse que este script se puede optimizar pero no es el objetivo principal de este proyecto.

```
function x = convolution(G,L)

%

% x = convolution(G,L)

%

% G es la 1° convolución de 3x3,

% L es la 2° convolución de 3x3

% OUTPUT. La matriz 5x5 equivalente de aplicar ambas convoluciones


%Se inicializa la matriz salida

x=zeros(5,5);

%Se empieza a llenar la matriz de resultados...

x(1,1)=L(1,1)*G(1,1);

x(1,2)=L(1,1)*G(1,2)+L(1,2)*G(1,1);

x(1,3)=L(1,1)*G(1,3)+L(1,2)*G(1,2)+L(1,3)*G(1,1);

x(1,4)=L(1,2)*G(1,3)+L(1,3)*G(1,2);

x(1,5)=L(1,3)*G(1,3);

x(2,1)=L(1,1)*G(2,1)+L(2,1)*G(1,1);

x(2,2)=L(1,1)*G(2,2)+L(1,2)*G(2,1)+L(2,1)*G(1,2)+L(2,2)*G(1,1);

x(2,3)=L(1,1)*G(2,3)+L(1,2)*G(2,2)+L(1,3)*G(2,1)+L(2,1)*G(1,3)+
      L(2,2)*G(1,2)+L(2,3)*G(1,1);

x(2,4)=L(1,2)*G(2,3)+L(1,3)*G(2,2)+L(2,2)*G(1,3)+L(2,3)*G(1,2);

x(2,5)=L(1,3)*G(2,3)+L(2,3)*G(1,3);

x(3,1)=L(1,1)*G(3,1)+L(2,1)*G(2,1)+L(3,1)*G(1,1);
```

$x(3,2) = L(1,1) * G(3,2) + L(1,2) * G(3,1) + L(2,1) * G(2,2) + L(2,2) * G(2,1) +$
 $L(3,1) * G(1,2) + L(3,2) * G(1,1) ;$

$x(3,3) = L(1,1) * G(3,3) + L(1,2) * G(3,2) + L(2,3) * G(2,1) + L(1,3) * G(3,1) +$
 $L(2,1) * G(2,3) + L(2,2) * G(2,2) + L(3,1) * G(1,3) + L(3,2) * G(1,2) +$
 $L(3,3) * G(1,1) ;$

$x(3,4) = L(1,2) * G(3,3) + L(1,3) * G(3,2) + L(2,2) * G(2,3) + L(2,3) * G(2,2) +$
 $L(3,2) * G(1,3) + L(3,3) * G(1,2) ;$

$x(3,5) = L(1,3) * G(3,3) + L(2,3) * G(2,3) + L(3,3) * G(1,3) ;$

$x(4,1) = L(2,1) * G(3,1) + L(3,1) * G(2,1) ;$

$x(4,2) = L(2,1) * G(3,2) + L(2,2) * G(3,1) + L(3,1) * G(2,2) + L(3,2) * G(2,1) ;$

$x(4,3) = L(2,1) * G(3,3) + L(2,2) * G(3,2) + L(2,3) * G(3,1) + L(3,1) * G(2,3) +$
 $L(3,2) * G(2,2) + L(3,3) * G(2,1) ;$

$x(4,4) = L(2,2) * G(3,3) + L(2,3) * G(3,2) + L(3,2) * G(2,3) + L(3,3) * G(2,2) ;$

$x(4,5) = L(2,3) * G(3,3) + L(3,3) * G(2,3) ;$

$x(5,1) = L(3,1) * G(3,1) ;$

$x(5,2) = L(3,1) * G(3,2) + L(3,2) * G(3,1) ;$

$x(5,3) = L(3,1) * G(3,3) + L(3,2) * G(3,2) + L(3,3) * G(3,1) ;$

$x(5,4) = L(3,2) * G(3,3) + L(3,3) * G(3,2) ;$

$x(5,5) = L(3,3) * G(3,3) ;$

APÉNDICE C. CÓDIGO HOST

```
/"This software contains source code provided by NVIDIA Corporation."

// includes, system

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <math.h>


// includes, project

#include <cutil.h>


// includes, kernels

#include <simpleTexture_kernel.cu>


//Variables:

//*****

char *image_filename = "izquierda.pgm";

char *image_filename2 = "derecha.pgm";

char *output_name = "SALIDA_IZQUIERDA.pgm";

char *output_name2 = "SALIDA_DERECHA.pgm";

//*****

//declaracion de prototipo de funciones

void runTest( int argc, char** argv);


////////////////////////////////////

// Programa main

////////////////////////////////////
```

```

int main( int argc, char** argv)

{

    runTest( argc, argv);

    CUT_EXIT(argc, argv);

}

////////////////////////////////////

// Funcion principal

////////////////////////////////////

void runTest( int argc, char** argv)

{

    CUT_DEVICE_INIT(argc, argv);

    // Se cargan ambas imagenes del disco

    unsigned char* h_data = NULL;

    unsigned char* h_data2=NULL;

    unsigned int width, height,width2,height2;

    char* image_path = cutFindFilePath(image_filename, argv[0]);

    char* image_path2 = cutFindFilePath(image_filename2, argv[0]);

    if (image_path == 0) {

        printf("Unable to source file file %s\n", image_filename);

        exit(EXIT_FAILURE);

    }

    if (image_path2 == 0) {

        printf("Unable to source file file %s\n", image_filename2);

        exit(EXIT_FAILURE);

    }

}

```

```

CUT_SAFE_CALL( cutLoadPGMub(image_path, &h_data, &width,
                             &height));

CUT_SAFE_CALL( cutLoadPGMub(image_path2, &h_data2, &width2,
                             &height2));


//Se chequea si las imágenes tienen el mismo tamaño...

if ((width!=width2)|| (height!=height2)){

    printf("Different Image sizes!!!!, check both and reload
           program, press a button to exit\n");

    scanf("%d",width);

    exit(EXIT_FAILURE);

}

//OJO,puede que se queden fuera de rango, podría ser necesario
//usar "long" o "long long"

unsigned int size = width * height * sizeof(unsigned char);

printf("Loaded '%s', %d x %d pixels\n", image_filename, width,
       height);

printf("Loaded '%s', %d x %d pixels\n", image_filename2, width,
       height);

//Se reserva memoria en el device para los resultados

unsigned char* d_data = NULL;

unsigned char* d_data2 = NULL;

static int *g_minSSD;

static int *g_minSSD2;

static size_t g_floatDispPitch;

CUDA_SAFE_CALL(cudaMallocPitch((void**)&d_data,
                               &g_floatDispPitch, width*sizeof(unsigned char),
                               height));

CUDA_SAFE_CALL(cudaMallocPitch((void**)&d_data2,
                               &g_floatDispPitch, width*sizeof(unsigned char),
                               height));

CUDA_SAFE_CALL(cudaMallocPitch((void**)&g_minSSD,
                               &g_floatDispPitch, width*sizeof(int), height));

```



```

CUDA_SAFE_CALL(cudaMallocPitch((void**)&g_minSSD2,
                               &g_floatDispPitch, width*sizeof(int), height));

g_floatDispPitch /= sizeof(float);

// allocate array y se copian las imagenes

cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(8, 0,
                                                           0, 0, cudaChannelFormatKindUnsigned);

cudaArray* cu_array;

cudaArray* cu_array2;

CUDA_SAFE_CALL( cudaMallocArray( &cu_array, &channelDesc, width,
                                  height ));

CUDA_SAFE_CALL( cudaMallocArray( &cu_array2, &channelDesc,
                                  width, height ));

CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array, 0, 0, h_data, size,
                                   cudaMemcpyHostToDevice));

CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array2, 0, 0, h_data2,
                                   size, cudaMemcpyHostToDevice));

// Se liga el array a la textura

CUDA_SAFE_CALL( cudaBindTextureToArray( leftTex, cu_array,
                                         channelDesc));

CUDA_SAFE_CALL( cudaBindTextureToArray( rightTex, cu_array2,
                                         channelDesc));

//Se crea el bloque de hilos para el kernel de la Laplaciana de
//la Gausiana, NOTA: es algo más rápido si es de 1D

dim3 dimBlock(ANCHO_BLOQUE,1,1);

//Se calcula el grid con el block definido anteriormente

dim3 dimGrid(divUp(width,ANCHO_BLOQUE),height,1);

//Se crea el bloque de hilos para los kernels de calculo del
//mapa de disparidades

dim3 dimBlock2(BLOCK_W,1,1);

dim3 dimGrid2(divUp(width, BLOCK_W), divUp(height,
                                             ROWSperTHREAD), 1);

//Se crea e inicia el timer

CUDA_SAFE_CALL( cudaThreadSynchronize() );

unsigned int timer = 0;

```

```

CUT_SAFE_CALL( cutCreateTimer( &timer));

CUT_SAFE_CALL( cutStartTimer( timer));

//Llamada al primer kernel

laplacianagausiana<<< dimGrid, dimBlock, 0 >>>( d_data, d_data2,
                                                width, height,g_floatDispPitch);

//Sincronizamos por si acaso algun hilo no ha terminado y la
//textura no se graba correctamente

CUDA_SAFE_CALL( cudaThreadSynchronize() );

//Pasamos el resultado al cu_array que a su vez éste está ligado
//a la textura

CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array, 0, 0, d_data, size,
                                   cudaMemcpyDeviceToDevice));

CUDA_SAFE_CALL( cudaMemcpyToArray( cu_array2, 0, 0, d_data2,
                                   size, cudaMemcpyDeviceToDevice));

//Se ejecutan los kernels para el calculo de las disparidades

disparidadizda<<<dimGrid2,dimBlock2,SHARED_MEM_SIZE>>>(d_data,
                                                         g_minSSD,width,height,g_floatDispPitch);

disparidaddcha<<<dimGrid2,dimBlock2,SHARED_MEM_SIZE>>>(d_data2,
                                                         g_minSSD2,width,height,g_floatDispPitch);

// Se chequea si algun kernel ha generado un error

CUT_CHECK_ERROR("Kernel execution failed");

CUDA_SAFE_CALL( cudaThreadSynchronize() );

//Se desligan las texturas

cudaUnbindTexture(leftTex);

cudaUnbindTexture(rightTex);

//Se para el timer y se muestra el tiempo

CUT_SAFE_CALL( cutStopTimer( timer));

printf("Processing time: %f (ms)\n", cutGetTimerValue( timer));

printf("%.2f Mpixels/sec\n", (width*height*2 /(cutGetTimerValue(
    timer) / 1000.0f)) / 1e6);

CUT_SAFE_CALL( cutDeleteTimer( timer));

```

```
// Se reserva memoria en el host para que se puedan mostrar los
//resultados

unsigned char* h_odata = (unsigned char*) malloc(size);

unsigned char* h_odata2 = (unsigned char*) malloc(size);

// Copiamos los resultados del device al host

CUDA_SAFE_CALL( cudaMemcpy( h_odata, d_data, size,
                           cudaMemcpyDeviceToHost) );

CUDA_SAFE_CALL( cudaMemcpy( h_odata2, d_data2, size,
                           cudaMemcpyDeviceToHost) );

// Se escriben los resultados en ambos archivos de salida

char output_filename[1024],output_filename2[124];

strcpy(output_filename, image_path);

strcpy(output_filename2, image_path2);

strcpy(output_filename + strlen(image_path) -
        strlen(image_filename), output_name);

strcpy(output_filename2 + strlen(image_path2) -
        strlen(image_filename2), output_name2);

CUT_SAFE_CALL( cutSavePGMub( output_filename, h_odata, width,
                           height));

CUT_SAFE_CALL( cutSavePGMub( output_filename2, h_odata2, width,
                           height));

printf("Wrote '%s'\n", output_filename);

printf("Wrote '%s'\n", output_filename2);


// Se libera la memoria

CUDA_SAFE_CALL(cudaFree(d_data));

CUDA_SAFE_CALL(cudaFree(d_data2));

CUDA_SAFE_CALL(cudaFree(g_minSSD));

CUDA_SAFE_CALL(cudaFree(g_minSSD2));

CUDA_SAFE_CALL(cudaFreeArray(cu_array));

CUDA_SAFE_CALL(cudaFreeArray(cu_array2));

free(h_odata);
```

```
    free(h_odata2);  
  
    cutFree(image_path);  
  
    cutFree(image_path2);  
  
    CUT_EXIT(argc, argv);  
  
}
```

APÉNDICE D.CÓDIGO DEVICE

```
/*"This software contains source code provided by NVIDIA Corporation."

#ifndef _SIMPLETEXTURE_KERNEL_H_

#define _SIMPLETEXTURE_KERNEL_H_

#define SQ(a) (__mul24(a,a)) // Si se pone __mul24(a,a) se realiza
                             //SSD,si se pone abs(a) entonces SSA

int divUp(int a, int b)

{

    if(a%b == 0)

        return a/b;

    else

        return a/b + 1;

}

//Variables:

//*****

#define ROWSperTHREAD 40

#define BLOCK_W 64

#define ANCHO_BLOQUE 64

#define RADIUS_H 3

#define RADIUS_V 3

#define MIN_SSD 3186225

#define STEREO_MIND 0.0f

#define STEREO_MAXD 80.0f

#define SHARED_MEM_SIZE ((BLOCK_W + 2*RADIUS_H)*sizeof(int) )

#define VALOR_INICIAL 255

//*****
```

```
// Se declara la textura

texture<unsigned char, 2, cudaReadModeElementType> leftTex,rightTex;

/////////////////////////////////////////////////////////////////

//! KERNEL PARA EL CÁLCULO DE LA DISPARIDAD IZQUIERDA

/////////////////////////////////////////////////////////////////

__global__ void disparidadizda(unsigned char* izquierda,
                               int *disparityMinSSD,

                               int width,

                               int height,

                               size_t out_pitch)

{

    extern __shared__ int col_ssd[]; // Columna de memoria
                                     //compartida

    float d;      // valor de disparidad

    int diff;     // valor de la resta temporal

    int ssd;      // total SSD para la ventana de busqueda

    int x_tex;    // coordenadas de la textura

    int y_tex;

    int row;      // el valor de la fila actual

    int i;        // indice para bucles

    // se usan defines para calcular las coordenadas

    #define X (__mul24(blockIdx.x,BLOCK_W) + threadIdx.x)

    #define Y (__mul24(blockIdx.y,ROWSperTHREAD))

    // para los hilos leyendo los valores extra este es el offset

    // de la memoria compartida para grabar los resultados
```

```

int extra_read_val = 0;

if(threadIdx.x < (2*RADIUS_H)) extra_read_val =
    BLOCK_W+threadIdx.x;

// Se inicializa la memoria

if(X<width )

{

    for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)

    {

        //hay que pensar a que valor se inicializa

        izquierda[__mul24((Y+i),out_pitch)+X] =
            VALOR_INICIAL;

        disparityMinSSD[__mul24((Y+i),out_pitch)+X] =
            MIN_SSD;

    }

}

__syncthreads();


if( X < (width+2*RADIUS_H) && Y < height )

{

    x_tex = X - RADIUS_H;

    for(d = STEREO_MIND; d <= STEREO_MAXD; d++)

    {

        col_ssd[threadIdx.x] = 0;

        if(extra_read_val>0)    col_ssd[extra_read_val] = 0;


        // Se hacen las primeras filas


        y_tex = Y - RADIUS_V;
    }
}

```

```

for(i = 0; i <= 2*RADIUS_V; i++)

{

    diff = tex2D(leftTex,x_tex,y_tex) -
           tex2D(rightTex,x_tex-d,y_tex);

    col_ssd[threadIdx.x] += SQ(diff);

    if(extra_read_val > 0)

    {

        diff = tex2D(leftTex,x_tex+BLOCK_W,y_tex)
               -tex2D(rightTex,x_tex+ BLOCK_W- d,
                     y_tex);

        col_ssd[extra_read_val] += SQ(diff);

    }

    y_tex += 1;
}

__syncthreads();

// Se acumula el total

if(X < width && Y < height)

{

    ssd = 0;

    for(i = 0;i<=(2*RADIUS_H);i++)

    {

        ssd += col_ssd[i+threadIdx.x];

    }

    if( ssd < disparityMinSSD[__mul24(Y,out_pitch)
                               + X])

    {

```



```
        izquierda[__mul24(Y,out_pitch) + X] =  
            (unsigned char) ((d*255.0f)/STEREO_MAXD);  
  
        disparityMinSSD[Y*out_pitch + X] = ssd;  
  
    }  
  
}  
  
__syncthreads();  
  
// Se computa el resto de filas  
  
y_tex = Y - RADIUS_V; // Esta es la fila que se  
                      //quitará  
  
for(row = 1; row < ROWSperTHREAD &&  
    (row+Y < (height+RADIUS_V)); row++)  
{  
  
    // Se resta el valor de la primera fila de la suma de  
    //la columna  
  
    diff = tex2D(leftTex,x_tex,y_tex) -  
           tex2D(rightTex,x_tex-d,y_tex);  
  
    col_ssd[threadIdx.x] -= SQ(diff);  
  
    // Se suma el valor de la siguiente fila  
  
    diff = tex2D(leftTex,x_tex,y_tex + 2*RADIUS_V+1) -  
           tex2D(rightTex,x_tex-d,y_tex + 2*RADIUS_V+1);  
  
    col_ssd[threadIdx.x] += SQ(diff);  
  
    if(extra_read_val > 0)  
    {  
  
        diff = tex2D(leftTex,x_tex+BLOCK_W,y_tex) -  
               tex2D(rightTex,x_tex-d+BLOCK_W,y_tex);  
  
        col_ssd[extra_read_val] -= SQ(diff);  
    }  
}
```

```

        diff = tex2D(leftTex,x_tex+BLOCK_W,y_tex +
                    2*RADIUS_V+1) - tex2D(rightTex,x_tex-
                    d+BLOCK_W,y_tex + 2*RADIUS_V+1);

        col_ssd[extra_read_val] += SQ(diff);

    }

    y_tex += 1;

__syncthreads();

if(X<width && (Y+row) < height)
{
    ssd = 0;

    for(i = 0;i<=(2*RADIUS_H);i++)
    {
        ssd += col_ssd[i+threadIdx.x];
    }

    if(ssd < disparityMinSSD[__mul24(Y+row,
        out_pitch) + X])
    {
        izquierda[__mul24(Y+row,out_pitch) + X] =
            (unsigned char)((d*255.0f)/STEREO_MAXD);

        disparityMinSSD[__mul24(Y+row,out_pitch)
            + X] = ssd;
    }
}

__syncthreads(); // Se espera a que termine todo
}
}

```

```

    }

}

/////////////////////////////////////////////////////////////////

//! KERNEL PARA EL CÁLCULO DE LA DISPARIDAD DERECHA

/////////////////////////////////////////////////////////////////

__global__ void disparidaddcha(unsigned char* derecha,    // Puntero a
                             //la memoria de salida para el mapa de disparidades

                             int *disparityMinSSD,

                             int width,

                             int height,

                             size_t out_pitch)

    // el pitch (en pixeles) de la memoria de salida para el
    //mapa de disparidades

{

    extern __shared__ int col_ssd[]; // Columna de memoria
                                    //compartida

    float d;    // valor de disparidad

    int diff;    // valor de la resta temporal

    int ssd;    // total SSD para la ventana de busqueda

    int x_tex;    // coordenadas de la textura

    int y_tex;

    int row;    // el valor de la fila actual

    int i;    // indice para bucles

    // se usan defines para calcular las coordenadas

    #define X (__mul24(blockIdx.x,BLOCK_W) + threadIdx.x)

    #define Y (__mul24(blockIdx.y,ROWSperTHREAD))

```

```

// para los hilos leyendo los valores extra este es el offset
// de la memoria compartida para grabar los resultados

int extra_read_val = 0;

if(threadIdx.x < (2*RADIUS_H)) extra_read_val =
    BLOCK_W+threadIdx.x;

// Se inicializa la memoria

if(X<width )

{
    for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
    {
        //hay que pensar a que valor se inicializa

        derecha[__mul24((Y+i),out_pitch)+X] = VALOR_INICIAL;

        disparityMinSSD[__mul24((Y+i),out_pitch)+X] =
            MIN_SSD;
    }
}

__syncthreads();


if( X < (width+2*RADIUS_H) && Y < height )
{
    x_tex = X - RADIUS_H;

    for(d = STEREO_MIND; d <= STEREO_MAXD; d++)
    {

        col_ssd[threadIdx.x] = 0;

        if(extra_read_val>0)    col_ssd[extra_read_val] = 0;
    }
}

```

```

// Se hacen las primeras filas

y_tex = Y - RADIUS_V;

for(i = 0; i <= 2*RADIUS_V; i++)

{

    diff = tex2D(rightTex,x_tex,y_tex) -
           tex2D(leftTex,x_tex+d,y_tex);

    col_ssd[threadIdx.x] += SQ(diff);

    if(extra_read_val > 0)

    {

        diff = tex2D(rightTex,x_tex+BLOCK_W,y_tex) -
               tex2D(leftTex,x_tex+BLOCK_W+d,y_tex);

        col_ssd[extra_read_val] += SQ(diff);

    }

    y_tex += 1;

}

__syncthreads();

// Se acumula el total

if(X < width && Y < height)

{

    ssd = 0;

    for(i = 0;i<=(2*RADIUS_H);i++)

    {

        ssd += col_ssd[i+threadIdx.x];

    }

}

```

```
    if( ssd < disparityMinSSD[__mul24(Y,out_pitch) + X])

    {

        derecha[__mul24(Y,out_pitch) + X]= (unsigned
            char) ((d*255.0f)/STEREO_MAXD);

        disparityMinSSD[Y*out_pitch + X] = ssd;

    }

}

__syncthreads();

// Se computa el resto de filas
y_tex = Y - RADIUS_V; // Esta es la fila que se quitará

for(row = 1; row < ROWSperTHREAD && (row+Y < (height+
    RADIUS_V)); row++)

{

    // Se resta el valor de la primera fila de la suma de
    //la columna

    diff = tex2D(rightTex,x_tex,y_tex) -
        tex2D(leftTex,x_tex+d,y_tex);

    col_ssd[threadIdx.x] -= SQ(diff);

    // Se suma el valor de la siguiente fila

    diff = tex2D(rightTex,x_tex,y_tex + 2*RADIUS_V+1) -
        tex2D(leftTex,x_tex+d,y_tex + 2*RADIUS_V+1);

    col_ssd[threadIdx.x] += SQ(diff);

    if(extra_read_val > 0)

    {
```

```

diff = tex2D(rightTex,x_tex+BLOCK_W,y_tex) -
      tex2D(leftTex,x_tex+d+BLOCK_W,y_tex);

col_ssd[extra_read_val] -= SQ(diff);

diff = tex2D(rightTex,x_tex+BLOCK_W,y_tex +
              2*RADIUS_V+1) - tex2D(leftTex,x_tex+d+
              BLOCK_W,y_tex + 2*RADIUS_V+1);

col_ssd[extra_read_val] += SQ(diff);

}

y_tex += 1;

__syncthreads();

if(X<width && (Y+row) < height)
{
    ssd = 0;

    for(i = 0;i<=(2*RADIUS_H);i++)
    {
        ssd += col_ssd[i+threadIdx.x];
    }

    if(ssd < disparityMinSSD[__mul24(Y+row,
                                     out_pitch) + X])
    {
        derecha[__mul24(Y+row,out_pitch) + X] =
            (unsigned char)((d*255.0f)/STEREO_MAXD);

        disparityMinSSD[__mul24(Y+row,out_pitch)
                        + X] = ssd;
    }
}

```

```

        }

    }

    __syncthreads(); // Se espera a que termine todo

}

}

}

}

}

////////////////////////////////////

//! KERNEL PARA EL CÁLCULO DE LA LAPLACIANA DE LA GAUSIANA

////////////////////////////////////

__global__ void

laplacianagausiana( unsigned char* izquierda, unsigned char* derecha,
                    int width,int height,size_t out_pitch)

{

    float suma = 0,suma2 = 0;

    int x = blockIdx.x*ANCHO_BLOQUE+threadIdx.x;

    int y = blockIdx.y+threadIdx.y;

    if(x < width && y < height)

    {

        //Se calcula el valor nuevo de cada pixel despues de pasar
        //la convolucion

        suma = -4*(tex2D(leftTex,x,y-2)+tex2D(leftTex,x-1,y-1)+
                    tex2D(leftTex,x+1,y-1)+tex2D(leftTex,x-2,y)+
                    tex2D(leftTex,x+2,y)+tex2D(leftTex,x-1,y+1)+
                    tex2D(leftTex,x+1,y+1)+tex2D(leftTex,x,y+2))-1*
                    (tex2D(leftTex,x-1,y-2)+tex2D(leftTex,x+1,y-2)+
                    tex2D(leftTex,x-2,y-1)+tex2D(leftTex,x+2,y-1)+
                    tex2D(leftTex,x-2,y+1)+tex2D(leftTex,x+2,y+1)+
                    tex2D(leftTex,x-1,y+2)+tex2D(leftTex,x+1,y+2))+
                    2*(tex2D(leftTex,x,y-1)+tex2D(leftTex,x-1,y)+
                    tex2D(leftTex,x+1,y)+tex2D(leftTex,x,y+1))+
                    32*tex2D(leftTex,x,y);
    }
}

```



```

        suma2 = -4*(tex2D(rightTex,x,y-2)+tex2D(rightTex,x-1,y-1)+
            tex2D(rightTex,x+1,y-1)+tex2D(rightTex,x-2,y)+
            tex2D(rightTex,x+2,y)+tex2D(rightTex,x-1,y+1)+
            tex2D(rightTex,x+1,y+1)+tex2D(rightTex,x,y+2))-1*
            (tex2D(rightTex,x-1,y-2)+tex2D(rightTex,x+1,y-2)+
            tex2D(rightTex,x-2,y-1)+tex2D(rightTex,x+2,y-1)+
            tex2D(rightTex,x-2,y+1)+tex2D(rightTex,x+2,y+1)+
            tex2D(rightTex,x-1,y+2)+tex2D(rightTex,x+1,y+2))+
            2*(tex2D(rightTex,x,y-1)+tex2D(rightTex,x-1,y)+
            tex2D(rightTex,x+1,y)+tex2D(rightTex,x,y+1))+
            32*tex2D(rightTex,x,y);

    izquierda[__mul24(y,out_pitch)+x] = (unsigned char)
                                         (suma/80.0f+128.0f);

    derecha[__mul24(y,out_pitch)+x] = (unsigned char)
                                       (suma2/80.0f+128.0f);

}

}

#endif // #ifndef _SIMPLETEXTURE_KERNEL_H_

```

APÉNDICE E. CARACTERÍSTICAS DE LA GPU

En este apartado se van a detallar las características de la GPU utilizada en todas las pruebas que se realizaron durante el proyecto.

Modelo: NVIDIA® GeForce® 8600M GT

Número de multiprocesadores: 4 (1 Multiprocesador = 8 procesadores).

Compute Capability: 1.1

- El número máximo de hilos por bloque es 512.
- Los tamaños máximos de las dimensiones x, y, z de un bloque de hilos son respectivamente 512, 512, 64.
- El tamaño máximo de cada dimensión del grid de bloques de hilos es 65535.
- El *warp size* es de 32 hilos.
- El número de registros por multiprocesador es 8192.
- La cantidad de memoria compartida disponible por multiprocesador es 16 KB organizada en 16 bancos.
- La cantidad total de memoria constante es 64 KB.
- La cantidad total de memoria local por cada hilo es 16 KB.
- El cache de trabajo para la memoria constante es de 8 KB por multiprocesador.
- El cache de trabajo para la memoria de texturas varía entre 6 y 8 KB por multiprocesador.
- El número máximo de bloques activos por multiprocesador es 8.
- El número máximo de *warps* activos por multiprocesador es 24.
- El número máximo de hilos activos por multiprocesador es 768.
- Para una textura unidimensional destinada a un array CUDA, el ancho máximo es de 2^{13} .
- Para una textura unidimensional destinada a memoria lineal, el ancho máximo es de 2^{27} .

- Para una textura bidimensional destinada a memoria lineal o a un array CUDA, el ancho máximo es de 2^{16} y la altura máxima 2^{15} .
- Para una textura tridimensional destinada a un array CUDA, el ancho máximo es de 2^{11} , la altura máxima es de 2^{11} y la profundidad máxima 2^{11} .
- El límite sobre el tamaño del kernel es de 2 millones de instrucciones de microcódigo.
- Soporta funciones atómicas operando sobre palabras de 32 bits en memoria global.

REFERENCIAS BIBLIOGRÁFICAS

- [1] Stam, J. Stereo Imaging with CUDA, January 2008.
- [2] Armingol, J. M. et al. IVVI: Intelligent vehicle based on visual information. Robotics and Autonomous Systems, Vol. 55, Issue 12, pp. 904-916, 2007. ISSN 0921-8890.
- [3] De la Escalera, A. Algoritmos de obtención de mapas densos de distancias en tiempo real mediante un sistema de visión estéreo, 2007.
- [4] De la Escalera, A. Visión por Computador: Fundamentos y Métodos. Prentice Hall, 2001.
- [5] Konolige, K. Small vision systems: Hardware and Implementation. Eighth International Symposium on Robotics Research, Hayama, Japan, October 1997.
- [6] Pérez, N. Apuntes del curso: Algoritmos de Estimación de la Geometría de Múltiples Vistas. Instituto de Ingeniería Eléctrica (IIE). Universidad de la República, Montevideo, Uruguay, 2002.
- [7] De la Escalera, A. Rectificación de imágenes estéreo, 2006.
- [8] Ayache, N. Artificial Vision for Mobile Robots: Stereo Vision and Multisensory Perception. The MIT Press, 1991.
- [9] Bouquet, J. Camera Calibration Toolbox.
<http://www.vision.caltech.edu/bouquetj/calib_doc/> [Consulta: 15 de noviembre de 2009].
- [10] Zhang, Z. Flexible Camera Calibration By Viewing a Plane From Unknown Orientations. International Conference on Computer Vision (ICCV'99), Corfu, Greece, pp. 666-673, September 1999.
- [11] Zhang, Z. A Flexible New Technique for Camera Calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 22, No. 11, pp. 1330-1334, 2000.

- [12] Sánchez, J. Generación de mapas de disparidad en un sistema de visión estéreo. Proyecto Fin de Carrera inédito. Escuela Politécnica Superior. Universidad Carlos III de Madrid. Leganés, 2004.
- [13] Kanade, T., Okutomi, M. A stereo matching algorithm with an adaptive window: theory and experiment. IEEE Trans. On PAMI, Vol. 16, No. 9, September 1994.
- [14] Zhang, Z. A stereovision system for a planetary rover: calibration, correlation, registration, and fusion. Proc. IEEE Workshop on Planetary rover technology and systems, 1996.
- [15] Krotkov, E., Hebert, M., Simmons, R. Stereo Perception and Dead reckoning for prototype lunar rover. Autonomous Robots, Vol. 2, No. 4, pp. 313-331, December 1995.
- [16] Fua, P. A parallel stereo algorithm that produces dense depth maps and preserves image features. Machine vision and Applications, Vol. 6, No. 1, 1993.
- [17] Bolles, R., Woodfill, J. Spatiotemporal consistency checking of passive range data. International Symposium on Robotics Research, 1993.
- [18] NVIDIA CUDA Programming Guide (version 1.0). Disponible en <<http://www.nvidia.es>> [Consulta: 1 de febrero de 2009].
- [19] NVIDIA Corporation. <<http://www.nvidia.es>> [Consulta: 10 de noviembre de 2009].
- [20] NVIDIA CUDA Programming Guide (version 1.0). Chap. 5, pp. 52-53.
- [21] Faugeras, O., Hotz, B. et al. Real time correlation-based stereo: algorithm, implementations and applications. INRIA. August 1993.
- [22] Scharstein, D., Pal, C. Learning conditional random fields for stereo. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), Minneapolis, MN, June 2007.
- [23] Chirinos, M., Atoche, J.R. Percepción de Profundidad: Apareamiento Estereoscópico. En: 7º Congreso Interamericano de Computación Aplicada a la Industria de Procesos (CAIP'2005), Vila Real, Portugal, 12-15 de Septiembre de 2005.
- [24] NVIDIA CUDA Best Practices Guide (CUDA Toolkit 2.3). July 2009.